DartSync Project Final Report

COSC 60 – Spring 2014

[“Hip”, “Hip"] – SVN /hip

Shuyang Fang

Ethan Yu

Charles Dong

Peter Saisi

**Table of Contents**

1. Project Description
2. Feature list
3. Design and Implementation
   1. Tracker
   2. Peer
   3. Peer-to-Peer Communication and Data Transfer
   4. File Monitoring
4. Applications
   1. iPhone
   2. Android
   3. Web
5. Lessons Learned
6. Appendix

**Project Description**

DartSync is a lightweight, distributed P2P version of Dropbox sync-and-share with local file exchange and synchronization across iOS, Android, Mac, and Linux devices. Instead of relying on a centralized cloud repository, files are exchanged locally between devices. DartSync is implemented as an always-on server that keeps track of a global file list, and a set of personal devices as peers.

**Feature List**

Core Project

* Synchronization of multiple files in a single directory
* Compatibility across Linux systems
* TCP connection and data transfer
* Local file monitoring
* Fetching data from multiple peers

Project Extensions

* Recursive files monitoring and synchronization of a directory tree
  + Piece-wise data transfers between all live peers to allow large files
* C implementation on OS X and Linux
* iOS application in Objective-C
* Android application in Java
* Website analytics in Ruby on Rails
* Delta file updates (rather than file replacement when updating files)
* Compression and encryption of all communication
* Password authentication
* Partial version control
* Raspberry Pi tracker support

**Design and Implementation**

Tracker

**Overview**

The “tracker” is an always-on server that communicates with peer devices and maintains a global file list. It listens for new peers and creates handshake threads for each live peer (described in ‘Major Functions’ below).

When a peer first connects to the tracker and is authenticated against the tracker’s global password, the tracker sends it the most up-to-date global file list. The peer will check this global file list against its own local file list for differences and download / delete files accordingly. If the peer had made changes to a file whilst offline, it will first update to the most current global version, and then apply its own changes and send a corresponding file update to the tracker.

The tracker continually communicates with each peer to 1) make sure they are still ‘alive’ and 2) process file updates from each peer, update the global file list accordingly, and broadcast the updated list to all live peers.

**Major Functions**

=== handshake =======================================================

Description: A new handshake thread is created by main() for each new peer that joins. It continually listens for messages from its peer and responds accordingly.

Pseudo Code:

* while (receive ptp\_peer\_t from peer)
  + case AUTHENTICATION
    - Check incoming password attempt against global password
    - Respond with PASSWORD\_CORRECT or PASSWORD\_INCORRECT
  + case REGISTER
    - Respond with KEEP\_ALIVE interval and global file list
  + case KEEP\_ALIVE
    - Update last time stamp in tracker-side peer table for this peer
  + case FILE\_UPDATE
    - getDifferences() between global file list and incoming file list
    - Broadcast changes to all live peers if needed
  + case CLOSE (or heartbeat timeout)
    - Remove the peer from the tracker-side peer table
    - Remove peer’s IP from all files’ peer IP arrays
    - Broadcast changes to all live peers
    - Cancel corresponding handshake thread

====================================================================

=== heartbeatTimer ====================================================

Description: This thread monitors which peers are alive and updates the tracker-side peer table and global file list accordingly.

Pseudo Code:

* Check the tracker-side peer table's last time stamps to see if it has been more than KEEP\_ALIVE\_TIMEOUT\_PERIOD since any peers have sent a KEEP\_ALIVE segment
* If a peer has timed out:
  + Remove the peer from the tracker-side peer table
  + Remove peer’s IP from all files’ peer IP arrays
  + Broadcast changes to all live peers
  + Cancel corresponding handshake thread

====================================================================

**Relevant Data Structures**

/\* Tracker-side peer table \*/

typedef struct \_tracker\_side\_peer\_t {

struct \_tracker\_side\_peer\_t \*next;

char ip[IP\_LEN]; //IP address of the peer

uint64\_t last\_time\_stamp; //last time the peer sent a KEEP\_ALIVE segment

int32\_t sockfd; //sockfd used to communicate with peer

int32\_t authenticated; //whether or not this peer has been authenticated

pthread\_t handshakeThread; //the peer’s corresponding handshake thread

} tracker\_side\_peer\_t;

/\* Data structure for Peer-to-Tracker communication \*/

typedef struct segment\_peer {

int32\_t type; // packet type

int32\_t port; // listening port number in p2p

char reserved[RESERVED\_LEN]; // reserved space: 8 bytes

char peer\_ip[IP\_LEN]; // the ip address of the peer sending this packet

char \*file\_table; // client file table

} ptp\_peer\_t;

/\* Data structure for T2P communication \*/

typedef struct segment\_tracker {

int64\_t interval; // time interval peer should send alive messages

char \*file\_table; // tracker file list

} ptp\_tracker\_t;

Peer

**Overview**

The “peer” represents any user device that connects to DartSync. It communicates with the tracker to send and receive file list updates, monitors a local DartSync directory, creates a .DartSync backup directory with old versions of files, and communicates with other peer devices to download and upload files.

When a peer receives a global file list from the tracker, it will check this global file list against its own local file list for differences and download / delete files accordingly.

When the file monitoring thread detects a change in the peer’s local DartSync directory, the peer will update its local file list accordingly and send a file update message to the tracker, which will then update the global file list and broadcast the updated list to all live peers.

Peer-to-Peer communication, data transfer, and file monitoring will be covered in the next sections.

**Major Functions**

=== main ============================================================

Description: The main thread connects to the tracker, logs in, receives a global file list from the tracker, and crawls the local DartSync folder before starting threads that monitor files, listen for peer download requests, and communicate with the tracker.

Pseudo Code:

* Connect to tracker
* Get user input to log in to tracker (and proceed if successful)
* Send REGISTER packet to tracker
  + Receive global file list and heartbeat interval in response
  + Crawl local folder to create local file list
  + getDifferences() between the two file lists
  + If differences exist, download/delete files accordingly
* Crawl the local folder again
* Send the local file list to the tracker
* Start the following threads:
  + file\_monitor (monitors local files)
  + P2P\_listening (listens for download requests from other peers)
  + alive\_thread (sends alive messages to tracker to maintain alive status)
  + logging\_thread (logs various information for analytics website to display)
  + waitTracker (listens for file updates from the tracker)

====================================================================

=== waitTracker =======================================================

Description: This thread receives file updates from the tracker and updates the local file list accordingly.

Pseudo Code:

* while (receive ptp\_tracker\_t from tracker)
  + getDifferences between incoming global file list and local file list
  + Update the local file list to match the global file list (including version numbers)
  + Call applyDifferences() to download / delete files accordingly

====================================================================

=== applyDifferences ===================================================

Description: This function receives the file\_collection\_t containing the differences between the peer’s local file list and the global file list. It has a linked list of new files, one of modified files, and another with deleted files.

Pseudo Code:

* Iterate through list of deleted files and create a deletion thread for each one
* Join all delete threads
* Iterate through list of new files and create a new\_dl\_thread for each one
* Iterate through list of modified files and create a mod\_dl\_thread for each one

====================================================================

**Relevant Data Structures**

/\* structure that will hold that linked lists for various categories of altered files \*/

typedef struct file\_collection{

file\_node\_t\* deleted\_head; //deleted files

file\_node\_t\* new\_head; //new and modified files (incl. renamed/moved)

file\_node\_t\* modified\_head;

file\_node\_t\* renamed\_head;

file\_node\_t\* common\_files;

} file\_collection\_t;

/\* Peer-side peer table \*/

typedef struct \_peer\_side\_peer\_t {

char ip[IP\_LEN]; // Remote peer IP address, 16 bytes

char \*file\_name; // Current downloading file name

struct \_peer\_side\_peer\_t \*next; // Pointer to the next peer, linked list

uint64\_t checksum; // checksum of file currently being downloaded

} peer\_side\_peer\_t;

/\* Data structure for Peer-to-Tracker and Peer-to-Peer communication \*/

typedef struct segment\_peer {

int32\_t type; // packet type

int32\_t port; // listening port number in p2p

char reserved[RESERVED\_LEN]; // reserved space: 8 bytes

char peer\_ip[IP\_LEN]; // the ip address of the peer sending this packet

char \*file\_table; // client file table

} ptp\_peer\_t;

/\* Data structure for Tracker-to-Peer communication \*/

typedef struct segment\_tracker {

int64\_t interval; // time interval peer should send alive messages

char \*file\_table; // tracker file list

} ptp\_tracker\_t;

Peer-to-Peer Communication and Data Transfer

**Download Overview**

After comparing the received tracker file list against its own, the peer knows which files are to be deleted, which files are to be modified, and which files are to be added (new).

A separate master thread is started to handle each instance of each such case on the side of the downloading peer.

If a file is to be deleted, a master deletion thread is started for that file. This thread first attempts to copy the file to be deleted into our .DartSync backup directory. Then, it attempts to delete the file itself.

If a file is to be modified, a master modification thread is started for that file. This thread first attempts to copy the file to be modified into our .DartSync backup directory (to backup its current revision). This master thread also truncates the copy of the file in our DartSync directory to the size of the most recent revision. Then, this master thread checks the number of alive peers who have the most recent revision of this file. The master modification thread spawns a downloading thread that downloads from each alive peer.

If a new file is to be added, a master new file download thread is started for that file. This master thread first creates an empty version of the file in its current location in our DartSync directory and truncates it to the correct size. This master thread also checks the number of alive peers who have this new file. The master new file download thread then spawns a downloading thread that downloads from each alive peer.

The download thread simply makes a request to its assigned peer. This request includes the name of the file to be downloaded, where in the file it needs information starting from, the length of the information needed, and the version the downloader has. The download thread then receives a response back and applies the response to the file.

**Upload Overview**

On the uploading peer’s side, it receives a connection attempt from a downloading peer. The uploading peer then spawns an uploading thread to handle the downloading peer’s dl request. This uploading thread handles both modification requests and new download requests. In the event of a modification request, the uploading thread attempts to find the revision the downloader has in the uploading peer’s backup directory. If this revision exists, the uploading thread finds the delta between the downloader’s version of the file and the uploader’s version of the file. This delta is sent back with appropriate parameters to allow the downloader to apply it. In the event of a new file download request, the uploading thread simply sends the enter piece of the new file requested by the downloader to the downloader.

**Overview Data Transfer**

The actual transfer of information is handled through complete serialization of the data. That is, everything is transferred as a char array. On the sender’s side, this char array is first compressed and encrypted before being sent. On the receiver’s side, the received char array is first decrypted and uncompressed before being analyzed.

File monitoring will be covered in the next section.

**Major Functions**

**Download (peer.c):**

=== master\_delete\_thread ===============================================

Description: This thread receives the file\_node\_t pointer of the file to be deleted and actually carries out the deletion.

Pseudo Code:

* Create a backup copy of the file to be deleted in the .DartSync backup directory.
* Delete the file.

====================================================================

=== master\_mod\_dl\_thread ==============================================

Description: This thread receives the file\_node\_t pointer of the file to be modified and oversees the modification.

Pseudo Code:

* Create a backup copy of the file to be modified in the .DartSync backup directory.
* Truncate the file to its modified size.
* Determine the number of peers that have the most recent revision of the file and are alive.
* Calculate the piece length to request from each peer as a function of the file size and the number of alive peers.
* Spawn a P2P\_download\_thread for each alive peer.

====================================================================

=== master\_new\_dl\_thread ===============================================

Description: This thread receives the file\_node\_t pointer of the new file to be added and oversees the new file download.

Pseudo Code:

* Create a blank copy of the file in the DartSync root directory with fully qualified path established.
* Truncate the file to its supposed size.
* Determine the number of peers that have the file and are alive.
* Calculate the piece length to request from each peer as a function of the file size and the number of alive peers.
* Spawn a P2P\_download\_thread for each alive peer.

====================================================================

=== P2P\_download\_thread ===============================================

Description: This thread receives a dl\_arg pointer that holds the information regarding the download itself. This thread then initiates the download and applies any changes.

Pseudo Code:

* Send the dl\_request contained in the dl\_arg through the port contained in the dl\_arg.
* Receive a linked list ul\_response from the uploader.
* Go through the ul\_response linked list, write the diff contained in each node starting at the start\_pos parameter in each node to the file contained in the dl\_request.

====================================================================

**Upload (peer.c):**

=== P2P\_download\_thread ===============================================

Description: This thread receives the file descriptor for the socket of the downloading peer. This thread then waits for the downloading peer’s download requests and attempts to service it.

Pseudo Code:

* Receive a dl\_request through the passed in port.
* Open the requested file, grab the entire piece starting at the requested start\_pos with requested length. Store this piece in a char buffer.
* If the request is for a new file, send the entire piece as a single ul\_response node with start\_pos equal to the requested start\_pos, length equal to the requested length, and diff equal to the char buffer data.
* If the request is for a modification:
  + Attempt to open the requester’s file revision.
  + Attempt to grab the same piece from the old revision.
  + Find consecutive differences between the old revision’s piece and the current revision’s piece. Construct a linked list of ul\_response nodes out of these consecutive differences.
  + Send this linked list of deltas to the downloader peer.

====================================================================

**Data Transfer (utils/socket\_common.c):**

=== send\_struct =======================================================

Description: This function receives the file descriptor for the socket to send through, the structure to be sent, and its type. This function handles the correct sending of the struct in a compressed, encrypted format.

Pseudo Code:

* Based on the structure type, serialize it accordingly, making sure to put things in network byte order.
* Compress the serialized struct. Encrypt it.
* Send the uncompressed length, the compressed length, and the encrypted length.
* Then send the actual payload itself in at most 1024 byte chunks.

====================================================================

=== recv\_struct =======================================================

Description: This function receives the file descriptor for the socket to receive through, the structure to be received, and its type. This function handles the correct receiving of a compressed, encrypted, and serialized struct and its subsequent decryption, decompression, and deserialization.

Pseudo Code:

* Receive the uncompressed length, the compressed length, the encrypted length, and the payload.
* Decrypt the payload. Decompress the payload.
* Finally, deserialize the payload.

====================================================================

**Relevant Data Structures**

/\* Structure that a download peer will send to the upload peer. \*/

typedef struct dl\_request {

char \*file\_name; // file the dler wants

uint64\_t start\_pos; // beginning to start calculating diff

uint64\_t length; // length to check

uint64\_t req\_ver; // version the dler has

char peer\_ip[IP\_LEN]; // IP of the upload peer

} dl\_req\_t;

/\* Structure passed to download thread. \*/

typedef struct dl\_arg {

dl\_req\_t \*dl\_req; // the dl\_req

uint64\_t new\_checksum; // the checksum of the ul

int32\_t port; // the port of the upload peer

} dl\_arg\_t;

/\* Structure that the upload peer will send to the download peer. \*/

typedef struct ul\_response {

uint64\_t start\_pos; // the beginning pos of the diff

uint64\_t length; // the length of the diff

char \*diff; // the diff

struct ul\_response \*next; // the next diff sequence

} ul\_resp\_t;

File Monitoring

**Overview**

The File Monitor is primarily tasked with monitoring the local file system to determine whether changes in the local directory have occurred. A change in the file system triggers the file monitor to send an updated global playlist to the tracker. The File Monitor is platform-independent and works on both OS X and Linux because it shuns global hooks, instead relying on a periodic traversal of the file system.

File Monitor is a misleading moniker, however, as it does more than just monitor the DartSync folder. The File Monitor implemented in lukefilewalker.c is tasked with:

* Monitoring the local file system (this incudes recursive monitoring)
* Version control
* Maintaining and updating the global file list
* Notifying the tracker when changes occur in the file system
* De/serializing of converted global file lists

**Major Functions**

======== monitor\_files ================================================

Description: crawls files every second and sends a FILE\_UPDATE segment to tracker if appropriate

Pseudocode:

* Initially, populate null globalfilelist with a linked list of existing files and directories
* Periodically recrawl the home directory
* Save current configuration information for directory including version numbers to a hidden folder called .SUPER\_DUPER\_SECRET\_STORAGE
* If recrawled linked list differs from current globalfilelist:
  + Update globalfilelist
  + Send globalfilelist to tracker
* Else sleep then recrawl again

====================================================================

======= getDifferences =================================================

Description: Compare the global file list and received file list to determine which files are new, deleted, and modified/renamed. Creates a file\_collection\_t struct pointing to disjunct linked lists of each of these types of files

Pseudocode:

* Obtain new and modified files by looking at the checksums and filenames that exist in the received linked list but not in our global linked list
* Partition the new and modified files into new and modified files, respectively
* Obtain the deleted (in global file list and not received list) files, which include renamed files
* Obtain common files between the two lists
* Convert the array of common files, deleted files, new files and modified files into separate linked lists
* Append each linked list to the collections structure whose five members each point to the head of each category of files.

This separation makes it easy for other external function calls to deal directly with the desired file category. For instance, the peer doesn’t have to concern itself with new files when it wants to delete files; it just obtains a handle to the deleted&renamed files.

====================================================================

**Version Control**

**Peer-Side:**

When the local file system is crawled, the file monitor updates the respective files’ version numbers using information stored in the hidden directory. If an entry for a file is found, it is assigned the version number found in that entry. If no entry is found, the file is assumed to be a new file and assigned a version number zero.

When the peer receives a file list from the tracker, it calls do\_version\_comparisons\_deleted, do\_version\_comparisons\_commmon, and do\_version\_comparisons\_modified. These functions ensure that we never delete or modify a file when we are the most up to date peer and apply the tracker’s version number updates.

**Tracker-Side:**

To solve the problem of offline version self-incrementing by a peer, only the server can give the go ahead to increase the version number. As a consequence, our versioning mimics git/svn in the sense that when a peer first comes online, it first has to update to the version the server has before layering on the changes that were made offline (think of it as ‘update’ then ‘commit’).

Version control on the tracker side is done by accessing the same three functions, which ensure that the tracker only modifies files submitted by the most up to date peer.

======= do\_version\_comparisons\_modified =================================

Description: removes modified files that have lower version numbers from linked list

Pseudocode:

For every file in linked list,

* If file version number < our version number
  + Remove from list
* Else
  + If tracker and version number > our version number:
    - Update our file

====================================================================

======= do\_version\_comparisons\_common ==================================

Description: Bumps up our global file list’s version numbers to current global level

Pseudocode:

* For every file in the linked list:
  + If our number < file version number, set our version number to file version number

====================================================================

======== do\_version\_comparisons\_deleted ==================================

Description: Ignores deletion requests for files that are ahead of tracker version

For every file in linked list,

* If file version number < our version number:
  + Remove file from deletion list
* If file current number >= our version number:
  + Remove file from global file list

====================================================================

**Relevant Data Structures**

typedef struct file\_name {

char name[WALK\_SIZE]; // size of fileName and filePath

struct file\_name\* next; // next directory

struct file\_name\* previous; // prev directory

} file\_name\_t;

typedef struct file\_node{

char\* name; // file name

char\*\* newpeerip; // for peer, peer IP address

uint32\_t numPeers; // # of peers who have the latest version of file

uint32\_t type; // file or directory

uint32\_t size; // size of the file

uint32\_t version\_number; // version control

file\_name\_t\* name\_pointer;

struct file\_node\* next;

struct file\_node\* previous;

uint64\_t checksum;

} file\_node\_t;

/\* structure that will hold that linked lists for various categories of altered files \*/

typedef struct file\_collection{

file\_node\_t\* deleted\_head; //deleted files

file\_node\_t\* new\_head; //new and modified files (incl. renamed/moved)

file\_node\_t\* modified\_head;

file\_node\_t\* renamed\_head;

file\_node\_t\* common\_files;

} file\_collection\_t;

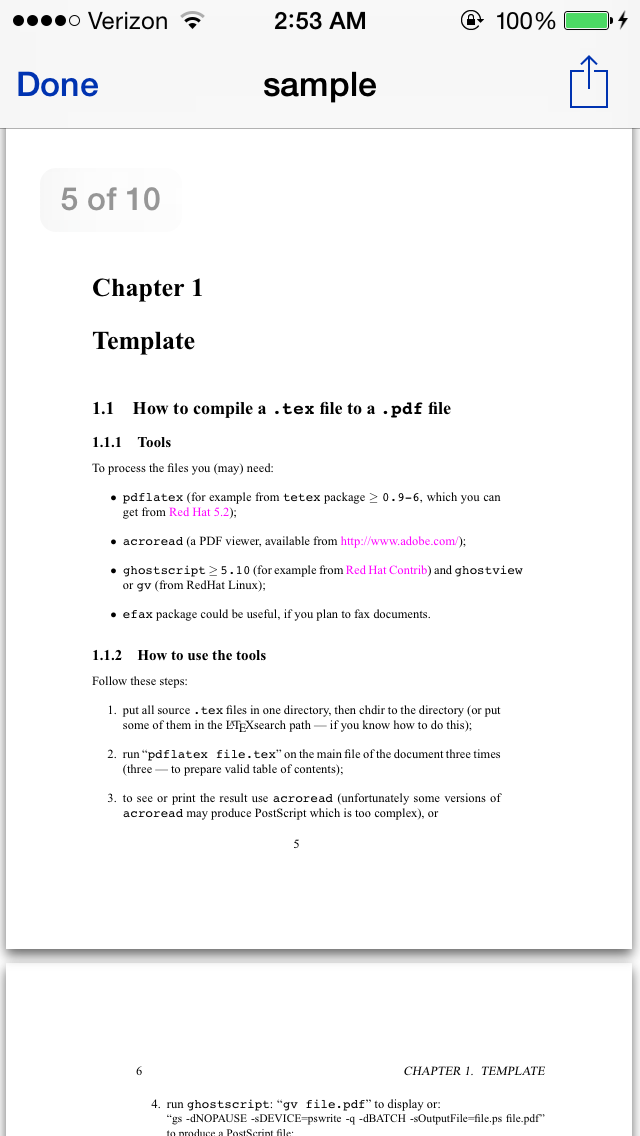
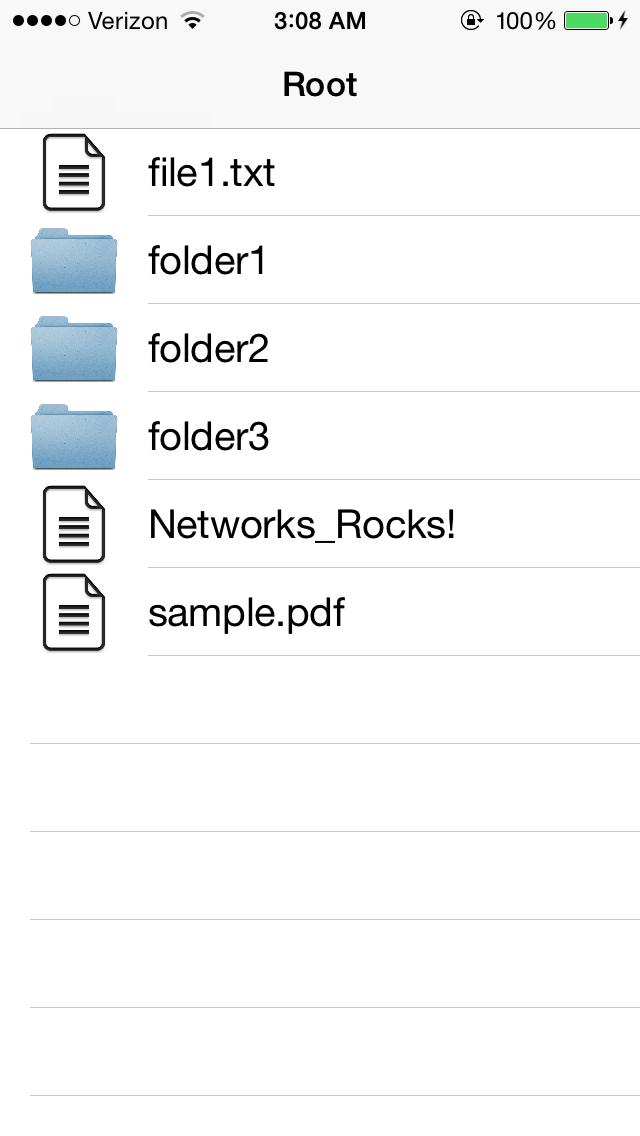
**Applications**

iPhone Application

The iPhone application, written in Objective-C and C, allows users to access their DartSync files from an iPhone. The iPhone runs a peer, obtains file updates from the tracker, and displays the directory tree in table format (Figure 1).

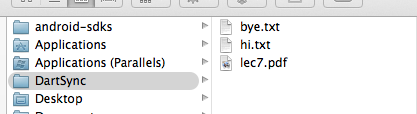
While the user cannot create or modify files from the iPhone, tapping a file in the file table causes the iPhone to download the designated file from a live peer and display it (Figure 2). If the file cannot be previewed directly inside the DartSync application, the user will be presented with alternative applications which may be able to preview the file (such as Dropbox or Pocket).

**Figure 1: Root Directory View Figure 2: Preview of sample.pdf**

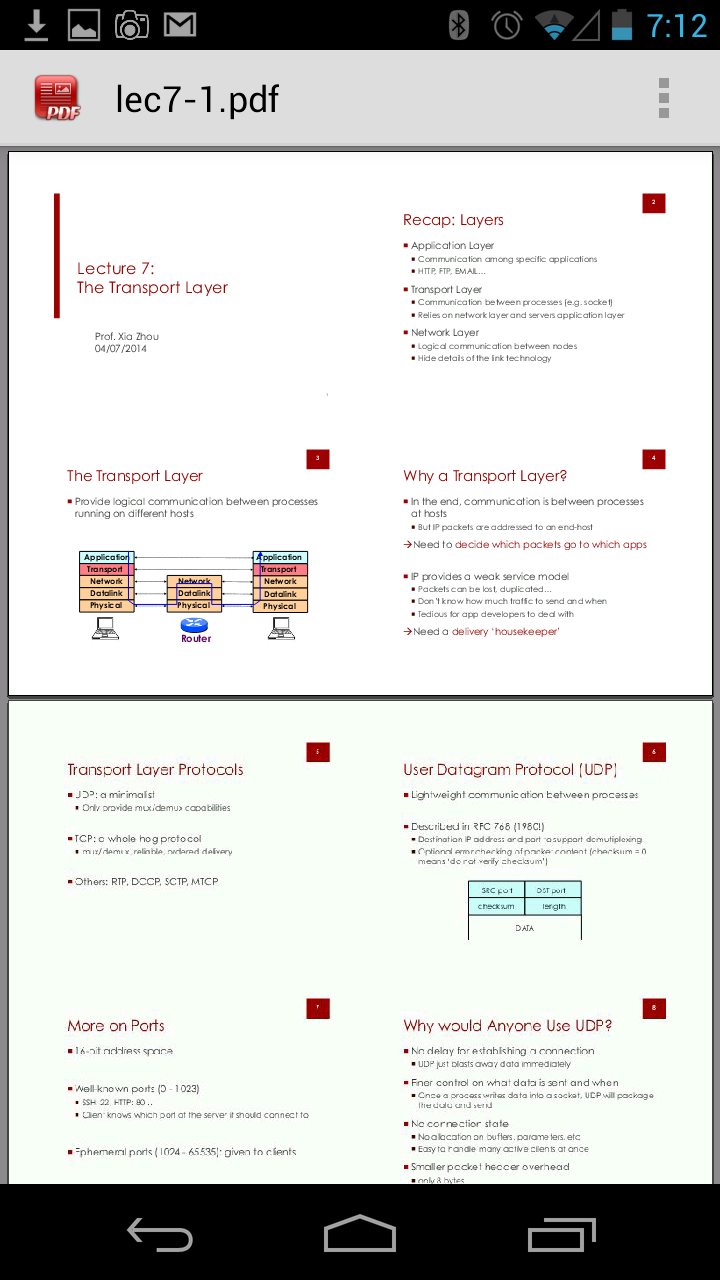
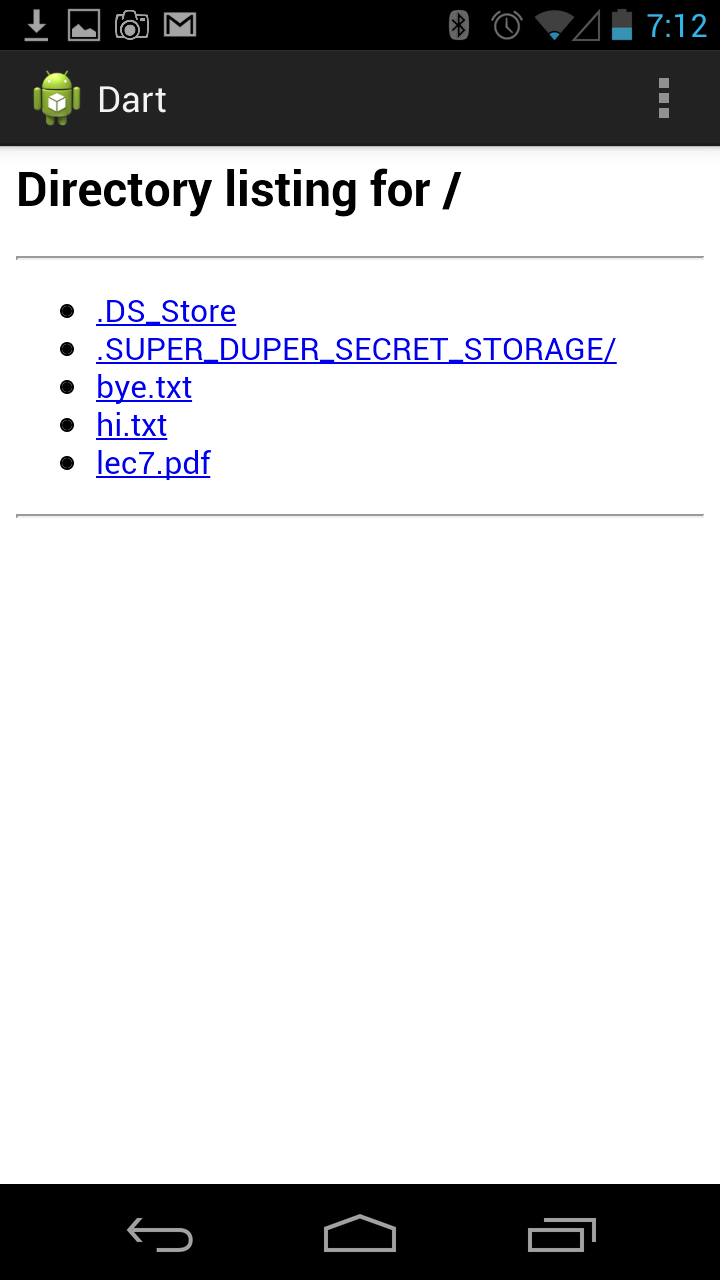


Android Application

**Figure 3: Sample DartSync Directory**



The Android application, written in Java, allows users to access their DartSync files from an Android phone running Android 4.0.3 and higher. The application connects to a peer, obtains file updates from the tracker, and displays the directory tree in table format (Figure 1).

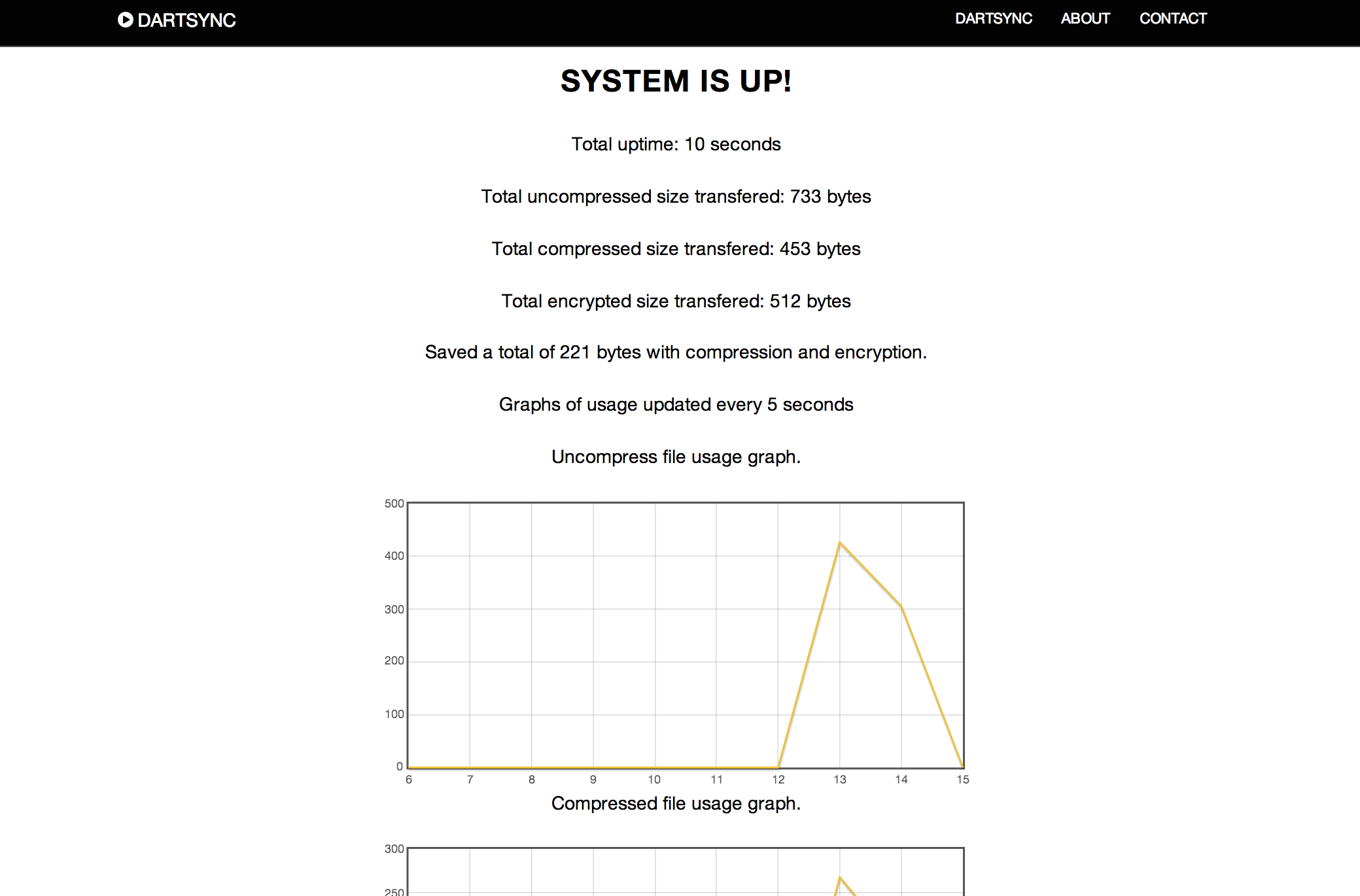
**Figure 1: Root Directory View Figure 2: Preview of sample.pdf** 

While the user cannot create or modify files from the phone, tapping a file in the file table causes the phone to download the designated file from a live peer and display it (Figure 2). If the file cannot be previewed directly inside the DartSync application, the user will be presented with alternative applications, which may be able to preview the file (such as Dropbox or Pocket).

Website Analytics

The web application, written in Ruby on Rails, provides analytics information to the user about the peer running on the system. It tells the user whether the local peer is online or offline, when the program was last started, how much data was transferred, graphics about network traffic, and information about the program and its authors. The web application utilizes web development technologies including but not limited to: HTML5, CSS3, JavaScript, JQuery, AJAX, SQLite, and the graphing library Flot.

The peer program periodically writes information to a hidden file, which the web application reads when queried for statistics. The server maintains a SQLite database with the values of all these parameters and updates them when they are requested. The server handles requests for URLS called Index, About, Contact, and Data. When the main webpage is first loaded, HTML and CSS display the Index page layout. It displays the system status and total downloaded file size, and has links to the About and Contact pages. Unbeknownst to the user, the JavaScript running on the website sends requests for the Data page in the background every five seconds, and the server responds to such requests with a JSON containing updated data. JQuery and AJAX work together to update elements on the website without needing to refresh the page, such as incrementing the number of bytes transferred, redrawing the graphs, updating the status of the peer.



**Lessons Learned**

Group Collaboration

This project not only taught us new technical skills and tools, we learned a lot about working together in a group. From the get-go, we realized that we had different areas of strengths and interests so delegating work appropriately became the crucial first step we needed to get right. Looking at the way we split up the work and how we progressed through the project, we learned that we probably should have put more thought into the delegation because that sets the course of the entire project. Understanding that one member enjoys tackling the difficult, tedious problems while another member can quickly produce lots of code that is simpler to conceptualize, and utilizing that information appropriately can make the collaboration process a lot more efficiently.

At times, our patience became strained and conflicts within the group sometimes got in the way of our personal relationships. Learning how to resolve these team conflicts was probably the most important lesson we learned from this project. In the future, we will encounter many conflicts at work, and if we do not understand how to collaborate with all types of coworkers, we will not succeed regardless of how well we can program.

Code Sharing

In the past, we have always been taught to comment our code extensively so that others can understand it. We always thought it was more of a suggestion than a rule. Whenever we looked at other people’s code, it was on assignments that we had also done ourselves, so it was easy for us to understand their code. But during this project other teammates wrote code for functions others did not design, so understanding their code was much more difficult. This became a problem especially during debugging stages, because it was very difficult for one member to debug another member’s code, so the members who wrote more lines of also had to spend more time debugging. This caused some members to have nothing to do while waiting for others to debug their code. This definitely could have been avoided if we wrote better documentation as we coded.

File System Monitoring

File monitoring in theory was straightforward to conceptualize and realize. The biggest lesson Saisi learnt was conflict resolution and versioning. Initially we used timestamps to determine whether a file the most up to date or recent, we soon saw the folly of this when we realized that some computers might have skewed clocks that will lead to incorrect versioning. Relying on versioning controlled by the server turned out to be the solution that solved the problem of having a peer go offline and come try to reconnect to its fellow peers after having done its own modifications. We chose the ‘update’ then ‘commit’ rout e whereby a peer has to first update to the most recent version of files before it can apply changes. In order to understand versioning correctly Saisi also came to study git and learn how it’s all just a graph with branches of different priority.

Data Exchange

Handling the serialization and deserialization of data was particularly challenging. Pointer logic and memory math required my utmost attention to detail and lead to many off-by-one error segfaults. This was also the first time I had to consider possible big-endian little-endian interactions. Making code that was multi-platform, multi-architecture compliant was a valuable lesson in not only attention to detail but also the importance of integration testing.

iOS Development

Over the course of this project, Charles learned how to write an iOS application from start to finish. It was a great opportunity for him to learn how to create a user interface, use the model-view-controller paradigm, and refactor/integrate code in two languages. This was his first time learning software development outside of Dartmouth classes, so he learned how to teach himself what was necessary to complete the task at hand in a limited amount of time.

Web Development

During the course of this project, Ethan learned about web development in Ruby on Rails. This was a great learning experience because it supplemented everything we learned in the course about the Internet, since websites are the most user-facing component of the Internet. This process taught Ethan about the basic tools involved in creating a dynamic website, such as HTML, CSS, JavaScript for the frontend, and Ruby on Rails and SQLite database in the backend. Although the challenge of integrating all of these moving parts was quite considerable, Ethan was able to put everything together and create a visually pleasant dynamic web application.

**Appendix**

Protocols

**AES Encryption**

AES is a new cryptographic algorithm that can be used to protect electronic data. Specifically, AES is an iterative, symmetric-key block cipher that can use keys of 128, 192, and 256 bits, and encrypts and decrypts data in blocks of 128 bits (16 bytes). Unlike public-key ciphers, which use a pair of keys, symmetric-key ciphers use the same key to encrypt and decrypt data. Encrypted data returned by block ciphers have the same number of bits that the input data had. Iterative ciphers use a loop structure that repeatedly performs permutations and substitutions of the input data. AES has been adopted by the U.S. government and is now used worldwide.

**zlib Compression**

Today, zlib is something of a de facto standard, with thousands of applications relying on it for compression, either directly or indirectly. These include: the Linux kernel, PNG, Apache HTTP server, OpenSSH, OpenSSL, Subversion, Git. The zlib DEFLATE algorithm combines elements of Huffman coding and LZ77 compression. Huffman Coding is used to prepare symbols for characters based on their frequency. LZ77 works by finding sequences of data that are repeated. Zlib combines elements of these two algorithms in a lightweight manner for a compression library that uses little resources and executes quickly.