[[1]](#footnote-1)

Designing a Dropbox-like File Storage Service with Client/Server Architecture

Dhruvin Patel

University of Mississippi

Oxford, MS

drpatel@go.olemiss.edu

*Abstract*—In this paper, I will present a reference of how I implemented a Dropbox-like prototype*.* Dropbox differs from the traditional models like server-client or peer-to-peer. It stores files on cloud-based computing and other features using their own cloud technology. I will explain how I designed my project using TCP and client and server model, which supports multiple clients. Also, I will explain what each file in the submission is for and how it is used. I will show the disadvantages of my implementation and fixes that could improve the performances.

*Keywords* – Dropbox, Server-Client, Performance

# INTRODUCTION

Dropbox has become a well-known application for storage and synchronizing file and folders. Dropbox provides file synchronization, user collaboration, and more to a massive number of internet users. Since Dropbox is powered by *cloud computing*, it can provide more than reliable file storage; however, we can create a Dropbox-like file storage using traditional server-client architecture. The inner working of Dropbox is not known because it is a company. However, people have design prototypes that can mimic the features that Dropbox already have.

Using server-client application to create a synchronization application, is much simpler to design; however, it takes away key features like user collaboration and backup deleted files. In addition to losing key features, the performance in my prototype is slower and has some overhead sending the file to a server and then to multiple clients.

In this paper, I will present the system architecture, protocol design, system implementation, performance, future works, submission files, and how to run and results. I will show how the server can handle multiple clients, and how clients can send and receive files at the same time.

# system architecture and Implementation

## Server

The server of my program acts as an Amazon S3 cloud storage and Amazon EC2 for computing and passing the file to clients. The server has to handle multiple clients to and synchronizes files and folder that the client and server may differ. In order to do this, my program creates a new thread for each client that is connected to the server. For each thread, the server handles if it needs to request or send files back or to the client.

In order to know which files to request and send, the server has to get the filenames and last modified dates of the files in a server’s directory that hold the synchronize files. After the client establishes a connection, the server receives a string with the filenames and the corresponding last modified longs. The server takes the string and splits it so that each file will but add to a HashMap. In the HashMap, the key is filename, and the value is the last modified long. Then the server creates another HashMap similar to the first but stores the server’s filenames and corresponding last modified longs.

The server first loops through the first HashMap checking if there is a file on the client but not on the server. If so the server creates a string and appends the filename to it. If the file is on both client and server and the last modified is higher on the client, we append the filename to the string.

Then the server iterates through the second HashMap that has the server’s filename and last modified values. In the loop, the server checks if a file in on the server but not on the client. If so, the server makes an Array List to add the filename to. It also checks if the file is on both server and client then the last modified date is higher on the server. Therefore, the server appends the filename to the Array List.

After creating a string of files to request and an Array List of files to send. The server sends the string to the client, in which the server waits for the client to send the number of files. The server then waits for all of the filenames from the client. Then the client sends the file sizes and last modified longs with the content of each file in which is saved in the server’s folder.

The server then sends the number of files to the client. Followed by the server will send the file names for all the files. The server then sends the file size, last modified, and content to the client for all the files. After this, the synchronizing process is finished, and the server and client files should match.

After the synchronizing process, the server creates a new thread that waits for a commend from the client. While waiting for the thread, starts WatchDir class which uses the *java.nio.file.WatchService* library.

The class detects changes in the folder that is synchronized to the server’s folder. If a file is modified or created, it outputs a one to the clients. Then the class sends the filename, last modified long, and the file size of the file to the clients. After, the class sends all file contents to the clients. If a file is deleted on the server, the WatchDir class will send a zero to the clients. Then the server sends the filename to the server so the clients know which file to delete.

The Wait class is a thread in the server that waits from respond from the client’s WatchDir. If the Wait class receives an integer one, then the Wait class know that a file has been modified or created in the client’s folder. The wait class then wait for the client to send the filename, last modified long, and file size. Then the server checks if the last modified long is greater than the file current in the server folder if so, the wait class overwrites the create file. If the last modified long is less than, then the file is not changed but is still read in to clear the input stream.

If the Wait class receives a zero integer, then it notifies the server to delete a file in the folder. The server gets the filename from the clients WatchDir class. The Wait class then checks if the file exists on the server’s folder, if so, the server will delete the file. The server will keep running the WatchDir and Wait class for each client until the client turns off the connection.

## Client

The client part of the program ensures that all files are uploaded to the server just like how Dropbox creates a folder that automatically synchronizes files. Multiple clients can be running the client code as long as the server is online and can accept the connection. The client part of the program works similar to that of the server but a few differences.

The program will first ask the user to input the path of the folder that is going to be synchronized. Then the client will establish a connection with the server using the server’s IP address and the port number. After making the connection with the server, the client then starts to synchronize the client’s files with the server files. The client first iterates through all the files in the folder and storing the filename and last modified in a HashMap. The key of the HashMap is the filename and the value stored is the last modified long value.

After the client saves all the files names in the HashMap, the client program will make a string, which will contain the filename and last modified date long separated by a semicolon and a comma in between each file. In order to do so, it iterates through the HashMap grabbing the key and the corresponding value.

After the string as all the filenames and last modified date, the client sends the string to the server though DataOutputStream from the socket. Then the client will receive a string from the server containing a list of the filenames that the server needs from the client. The client first splits the string to get the number of files that the server needs. If the number is less than or equal to zero. The client sends back a zero integer to the server to notice that is not sending any files. If the number is greater than zero, then the client returns the number of the file that needs to be sent.

The client then sends the filename, file size, and last modified date of the file to the server. For each file, the client then sends the file content to the server. Last, the client set the last modified date to the last modified long before the client read in the file.

The client will send all the files that were requested. The client then waits for the files that are missing on the client that the server has. The server sends an integer to the client so that the client will know how many file if any are going to be sent. If the number is greater than zero, the server will send one file at a time. The server first sends the filename, file size, and last modified date long in that order before the file content. The client creates a new file with the same filename. The content of the file is read in using a while loop until all the file as the file size as the file being sent from the server. Last the file’s last modified long is sent to the same long as the last modified long. After all the files are saved on to the client from the server, the sync process is completed.

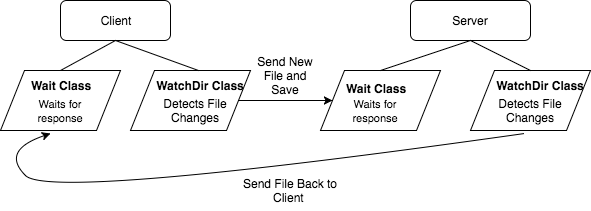
The client now creates a thread and start using the Wait class in the thread. This Wait class is the actual same as the server’s Wait class. The class waits for an integer to be received from the server. If that integer is a one, then the client knows that the server contains a file that is modified or created a new file. The client then receives the file’s filename and last modified date from the server. The client checks if the file as a last modified date greater than if the client’s file with the same filename. If so the server sends the file size and the file content to the client. The client then saves the file contents and updated the last modified date to the same as the server’s file last modified date. If the file’s last modified date is less than the client’s last modified date then the client gets the file content but does not save it in the file. If the integer is a zero, then the client knows that the server has delete a file. The server sends the client the filename and the client check if the file exists. If so, the client deletes the files.

While the Wait class is run, the client is also running the WatchDir class. This class is also the same as the server’s WatchDir class. The class detects changes in the folder that is synced to the client’s folder. If a file is modified or created, it outputs a one to the clients. Then the class sends the filename, last modified long, and the file size of the file to the clients. After, the class sends all file contents to the clients. If a file is deleted on the server, the WatchDir class will send a zero to the clients. Then the server sends the filename to the server so the clients know which file to delete.

The WatchDir and Wait class work simultaneously. The two class work together to update, create, and delete files from the client to server. These two class will continue to run throughout the program until the client or server quits the program.

## Design flow

In Figure 1, it demonstrates what happens when a new file is added to a client connected the server. After the initial synchronizing process, the WatchDir and Wait classes are both running on the client and server. Therefore, when a file is changed, created, or deleted the client’s WatchDir will send the file information to the server where the server’s Wait class will decide base on last modified date to save the file content or just ignore it. In case of a deleted file, the server will delete the file. If the server changes any file in the directory then the server’s WatchDir will output the file again to the client’s Wait class.



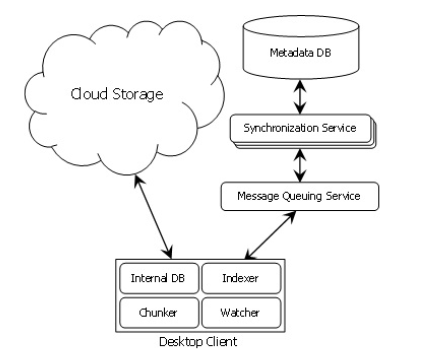
**Figure 1: Client/Server Design Flow**

## Comparison to Dropbox

My implementation is very different from that of how Dropbox implementation; however, my implementation does share some common architecture. Khanzadeh *et al.* [2] suggested that there are five main components to Dropbox-like cloud storage: “desktop client application, synchronization service, message queueing service, metadata database, and the cloud storage back-end.”

In my client-server design, the client has the components of desktop client application and synchronization. For example, the client can synchronize files whenever a modify, create, or delete happens in the client folder. The server has the synchronization service and cloud storage components. An example of this is when a client uploads a file it is then saved into the server’s folder and sent to other clients.

In a cloud computing design, the desktop client can be broken into watcher, chunker, indexer, and Internal DB [2]. The watcher monitors the folder and signals the program that a file as changed. The chunker splits the files into smaller pieces. Indexer records the events from Watcher and updates the database. Also, the internal database records tracks of the chucks and more [2]. In Figure 2, Khanzadeh *et al.* they presented how the logical architecture could be visualized.



**Figure 2. Logical Architecture [2]**

In my design using client-server architecture, I have used similar components in client-server architecture. Both the client and server share the same class for watcher and indexer. The WatchDir class watches the chosen folder to synchronize. It watches for events changes. For example, when a file is created, modified, or deleted, it will create a new event. The indexer of my program would be the Wait class. Once an event is created, it notifies the Wait class and processes the file according to the event. The internal database in my program would be the HashMap of filenames and their corresponding last modified long values. The synchronization service is on both the client and server as well. The client has to have the same file as the server, and the server has to have the same files as the client. The program ensures that all the files are the same by passing the files over TCP. All in all, my client-server program uses components that a cloud-based Dropbox would use.

# protocol design

The client and server architecture relies on TCP connection. TCP stands for Transmission Control Protocol. TCP provides reliable transport, flow control, and congestion control. TCP is needed in the project because the program is sent file content from the client to server and from the server to clients. Reliable transport is a need in order to make sure that the entirety of the file is transmitted.

Dropbox uses the TCP protocol to pass the information to the cloud, where it stores and computes differences in the file [2].

# performance evaluation

## Overhead

My version of Dropbox using client-server has overhead. First, the Watch Service java class has a delay to detect an event. Watch Service relays on the operating system to detect file changes. If multiple files are changed, Watch Service will process each event sequentially and wait for the next to process. In my development, the Watch Service can take up to five seconds to detect a new event.

Also, another overhead in my program is when there is a file that needs to be synchronized. An example of this is when a client creates or modifies a file. The client sends the whole file to the server. When the server receives the file, it creates the file in the server folder. This causes the server’s WatchDir to make a new event. This, in turn, causes the WatchDir to send out a one integer to all the connected clients. The server’s Watch Dir then, in turn, sends the file back to all the client. The server will send back the file to all the clients including the original client how created or modified the file. This cause the file to transfer multiple times. At most, a new file or modified file will be transmitted two times if there is only one client connected to the server. Since my program is not using a chunker to split the file and then update the chuck that is created, the program has to transmit the whole time every time a client creates or modifies a file. If the file size is larger, the program would take larger to synchronize.

## Performance

The program will synchronize all files in the folder and will take update any changes in the files within the folder. The program will throw an error if the client or server quit out of the application that anytime; however, if a client quits the server will still run for the other clients. The client is allowed to rejoin the server without any issues. Once a client joins the server it has to perform the initial synchronize between the client and server. The program also detects if a client has a file that has been modified and the server has an older version of the file. This helps the server have all files correct the latest version.

Depending on the number of files differences the synchronizing progress varies. In Table 1 and Table 2, I have tested my code to with multiple different file sizes and for a different number of users. The time it takes for one file to be sent from the client to server depends on the file size and number of clients connected to the server. As the file sizes increase so do the number of milliseconds to synchronize the file.

In Table 2, since the number of clients are two. The computation is for both the server to receive the file and the other client to receive the file. As the number of clients and file size increase so do the amount of time to sync the file. For example in Table 2, a 500 Kbs file take 71 milliseconds with two clients while the same file with one client takes 21 milliseconds.

One downside to the performance of my project is that it does not handle well with more than two clients or files larger than one hundred kilobytes. This is because for each client on the server the server creates a new thread, and within that thread, the server has both Wait and WatchDir class running. Therefore, each time the file is changed on the server the WatchDir responds to the number of threads active sending the file to all clients the number of open connection the server has.

Table 1

Per One Client

|  |  |
| --- | --- |
| Files Size | Time to Sync |
| 10 kb | 2.59 ms |
| 500 kb | 20.99 ms |
| 1000 kb | 22.22 ms |

Table 2

Per Two Client

|  |  |
| --- | --- |
| Files Size | Time to Sync |
| 10 kb | 5.03 ms |
| 500 kb | 70.85 ms |
| 1000 kb | 165.66 ms |

To improve these results, if I could find a way to only have one active WatchDir for all the connection the server has, it would reduce the number of times the server resends the file. Currently, each thread has an active WatchDir so the number of the clients increase the number of WatchDir, which therefore increase the number of time the file is sent out. It will reduce the number of times a file is transferred by half.

# future works

The future works of this project would use a hashing algorithm to for chucking a file and detecting file differences. Having a chunker would reduce the size of the file being sent over to the server and clients. The chunking of a file would require an internal database to control the number of chunks in a file. The client would have to update the database whenever a file is changed or updated. The updating and detecting change would cause more computation on the CPU; however, it will reduce the amount of data sent through the sockets.

Also, finding a way to reduce the number of times a file is sent to the server and clients would improve the performance of the synchronization. Currently, the server and client has to send the file back to the original source to verify that the is currently updated. Since the Wait function is waiting for an input from the server or client, it can only read in from the socket. And the WatchDir class will wait for a file to be updated so the class will only output to the socket. Currently, there is no way to have both classes to use an input and output stream at the same time without causing errors. Fixing this issue would reduce the amount of data passing through the socket.

Fix these two issues would result in better performance and fix redundancy in sending file content. The cost would reduce too since the code will only update a chunk of a file instead of the whole file and not send the whole file back to the client.

# submission files

The submission files include: Client.java, Server.java, Wait.java, WatchDir.java, ServerTest folder, Hello World.txt, test.txt, text2.txt, SampleTextFile\_10kb.txt, SampleTextFile\_500kb.txt, SampleTextFile\_1000kb.txt and Dropbox Paper.doc. Client.java is the client code for the project. Server.java is the server code for the project. Wait.java is the waiting class. WatchDir.java is the watch directory function of the project. Hello World.java, test.txt, test2.txt, SampleTextFile\_10kb.txt, SampleTextFile\_500kb.txt, and SampleTextFile\_1000kb.txt are the test files in the project. All the test files are in the ServerTest folder. Last is this report.

# how to run and results

In order to run the project, a user must have the Java SE 1.8 or higher. First, open the Server.java in a text editor, and update the path to the folder you want the sync with other clients. Then the code is ready to be compiled and run.

On Windows operating system, open a command line and go to the directory when the files are saved. Next, compile Server.java, Client.java, Wait.java, and WatchDir.java using “javac <filename>” for each file. After all, files are compiled, first, start the server using “java Server.java”. Then open a new command line prompt window and run the client code using “java Client.java”. The command line prompt will promote you to enter the path for a folder that will be used to sync. The user will have to enter the path. Repeat the last two steps for any number of users. After the server and client files are synced, the client can create files, updated current files, or delete files and the server will update the server files and any other connected clients. To quit out of the process hold ctrl and c.

On Mac or Linux, open a terminal and go to the directory of the folder. Next, compile Server.java, Client.java, Wait.java, and WatchDir.java using “javac <filename>” for each file. After all, files are complied, first start the server using “java Server.java”. Then open a new terminal window and run the client code using “java Client.java”. The terminal will promote you to enter the path for a folder that will be used to sync. The user will have to enter the path. Repeat the last two steps for any number of users. After the server and client files are synced, the client can create files, updated current files, or delete files and the server will update the server files and any other connected clients. To quit out of the process hold control and c.

The result will be the same on all operating systems. The client will ask for the folder to sync, then the server and client will start the initial syncing process. The server and client will output files that it is sending or receiving. The WatchDir will output if a file has been changed and the Wait class will output if a new file has been sent over or deleted.

# Conclusion

All in all, this paper covers how I used client-server architecture to implement a Dropbox-like file syncing service. The server acts as the cloud storage and pushing out file changes to the clients. The client gets the updated file from the server. By utilizing the java libraries like Watch Service and creating threads, my architecture can hold multiple clients with multiple files changes. By using server/client architecture, the project can handle updating or deleting files but loses some key features like user collaboration, back up files, and more. However, there are a few major drawbacks to how I implemented this Dropbox-like prototype. Copying over the whole file and redundant file transfer between server and client. There is a future implementation of this project that would improve the efficiency and reduce file transmissions.

References

1. H. Wang *et al,* “On the impact of virtualization on dropbox-like cloud file storage/synchronization services,” in 2012, . DOI:10.1109/lWQoS.2012.6245967.
2. Designing a Dropbox-like File Storage Service. [Online]. Available: http://www.academia.edu/12325670/Designing\_a\_Dropbox-like\_File\_Storage\_Service.

1. [↑](#footnote-ref-1)