

## A Cloud Storage System Using Concurrent Techniques

Akhil Kumar Harikumar

Samuel Smitherman

Christian Teeple

Professor Lei

CSE6324-002

## Introduction

A cloud storage system is one with many moving parts, requiring synchronized operations and communications between a client and server. The solutions for this type of system must be robust enough to mitigate environmental factors and user errors. Indeed, the goal of this assignment was to develop a real-world understanding of concurrency and its importance. With that said, a great deal of effort was spent to make the application as functional as possible. As such, the purpose of this document is to detail the design, the functionality, and testing of one such cloud storage system developed by its authors. In the later part, there will be a discussion focused on potential future works and the lessons learned during the course of the system's development life cycle.

## Design

The individuals reading this are more than likely already familiar with the client server architecture. As such, little time will be spent explaining this. To summarize, however, the system uses a cloud server architecture; as such, it consists of two programs: one for the server and another for the client. The server holds resources and its respective program manages these resources. The client is an application that a user can control to request access to and interact with the resources held by the server.

With this architecture in mind, the discourse surrounding the system's design will be broken into five sections. One to detail the implementation of the server. Another to detail the implementation of the client. The third one to detail the implementation of the communication between the two programs. In each of these facets of the system, there exists concurrency which will be thoroughly discussed in the fourth section. Finally, information will be provided on miscellaneous design items such as helper classes.

## Server

The resources held by the server are files which are stored in an SQLite database. In order to manage this database, the server program must be first initiated. Additionally, if there is no server active, the clients cannot do anything aside from watching the local directory—their primary purpose is to interact with the server to perform operations on the resources it holds. Once the server is initiated, it waits until it establishes a Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) connection with a client. Once it does this, it creates a new thread to manage interactions with that specific client.

This “Server Thread” will wait until it receives a TCP message from a client. There will be more details in the communications section, but the commands the Server Thread can receive are limited to upload and delete. If the operation is an upload, the server confirms that it is ready to begin. If it is a delete the server begins the process. (This is due to the simplicity of delete, again, which will be discussed later.)

In either case, the program creates an instance of two controller classes: a “Server Controller” and a “Data Controller”. These are Java monitors that will be used to synchronize data flow and protect the critical regions on the server. Additionally, a “Bounded Buffer” and a “Synchronizer” object are created as well. These will be used as control objects. These are passed to the next thread, the “Server Receiver”.

Based on the message received by the server, the Server Receiver will use the Server Controller to perform one of the two specified operations. If the message is for a delete command the Server Receiver will call a synchronized method in the Server Controller which will delete the data in SQLite database. If the message is for an upload, the Server Receiver will call set a boolean embedded into the Bounded Buffer to true, which is used to indicate that an upload is currently in progress. Additionally, it will call a synchronized method in the Server Controller which will instantiate several “Receive Threads”. The number of Receive Threads generated will vary based on information contained in the message.

These Receive Threads will wait to receive UDP packets from the client. After receiving the data packets, they are converted to byte arrays and deposited into the Bounded Buffer. The Bounded Buffer is passed into an instance of the “DBWriter” thread class. The DBWriter withdraws the data from the Bounded Buffer and assigns it to a jagged array data structure. Once the DBWriter threads receive all the packets (as indicated in the TCP message), it combines them into blocks before then merging them into a single byte array. This array is then used to write the data into the database.

While all this is occurring, the Server Receiver thread from earlier is still running in parallel. It is made to wait by the boolean value embedded in the Bounded Buffer noted above. Once the upload is complete, the Server Receiver thread will continue and check to see how many active clients there are. If there is only one active client, the Server Receiver finishes. Otherwise, it calls a method in the Server Controller to synchronize the uploaded date to each other active client (different from the one who made the received request). These will be referred to as additional clients.

The method that is called in the Server Controller creates a “DBReader” thread object. The DBReader calls a method in the Data Controller instantiated earlier depending on the operation that needs to be performed. If the operation is a delete, the Data Controller sends a message to one of the additional clients indicating that the client program should delete the file from its local directory. If the operation is a download (since the clients are now receiving data), the above process happens in reverse.

The Data Controller will first toggle to true a boolean inside the Synchronization object created earlier to indicate that data is being sent. Then the Data Controller will send a TCP message to the additional client indicating that data is soon to be sent. It then creates a variable number of “Send Threads” which will be determined by the data being sent. Before passing it to each Send Thread, the data is broken into blocks and again into packets. Each packet is deposited into the Bounded Buffer. Once it finishes the Data Controller toggles the boolean value in the

Synchronization object to false to indicate it is no longer sending anything. In the Send Threads the data is withdrawn from the Bounded Buffer and sent to the client.

## Client

The general flow of the client program is similar to the server. One of the main differences is that the user can directly interact with the client, suspending and resuming the communications with the client at the click of a button. Since the client must respond to a user's input, some time will be spent discussing how this was implemented.

The Java NIO API is an alternate API that can be used to perform operations on files. One particularly interesting feature is the Watcher Service, which can be used to watch a directory in the local file system. This watcher detects changes to files in the event of file creation, file deletion, or file modification. These events are generated by the user's operating system directly and are not managed by the system. This watcher is integrated into the cloud storage system discussed in this document.

While relegating these processes to the operating system, two unfortunate issues arose immediately. The first was that some operating systems generate two events when adding a file to a directory. The first event is the create event and the second is a modify event. The modify event is initiated because the timestamp is updated after the file is created. This resulted in multiple requests to upload the same file, resulting in the program breaking.

The second issue was that once the server sends a file to an additional client and the file gets written to the local directory, the watcher sees the creating event and begins to upload the file to the server. This results in an infinite upload loop between all the active clients which results in the programs crashing.

These issues were patched by using a Synchronization object. The Synchronization object holds a list which is used to block specific files from generating an upload or delete request. To do this, multiple instances of the Synchronization class were used. One was used to block uploads of files that were currently being processed. This was used to fix the multiple event generation bug. The other was used to block downloads of files that had not finished writing to the local directory. This was used to fix the infinite upload loop.

Another facet of the client is the pause and resume buttons. The user can click these buttons to temporarily stop in progress synchronization or resume said synchronization. To handle this functionality, two methods were added to the Synchronization objects. The first method is used such that if a boolean inside the Synchronization object is true, the threads sending data to the server or writing the file to the local directory wait. This boolean is set to true when the user clicks suspend. Additionally, the waiting threads are notified when the user clicks resume.

Outside of these user controls, the client works similar to the server (only in reverse). When the client is initiated, it establishes a TCP and UDP connection with the server. Once it does

this, it creates several objects for several classes, including a Bounded Buffer, an “Event Watcher” thread, a TCP based Receive Thread, and a “Client Controller”. Then the client begins waiting for an event to be seen by the watcher or to receive data from the client. The former will be explained first.

Once the user makes a change to the synchronized directory, the Event Watcher (which as the name suggests, uses the Watcher Service discussed above), instantiates a single “File Controller” and many “Event Processor” threads for each event seen. The single File Controller is a Java monitor and is shared amongst all the Event Processors. It is used to synchronize the operations performed on the files on the client side.

The Event Processors create a different instance of the “File Reader” thread based on the type of event that is processed. The File Readers calls synchronized methods in the File Controller to perform the operations. As in the case of the server’s DBReader and Data Controller, if the delete operation is performed it sends a TCP message to the server indicating the file should be deleted from the server.

If the upload operation is performed it sends a TCP message to the server to see if the server is ready to receive data. If it is, the File Controller sends a message to the client containing information about the data being send and what type of process needs to take place. It then segments the data into blocks and packets. Here the client checks to see if the pause button has been clicked; if not, it continues. It deposits each packet into the Bounded Buffer and passes the Bounded Buffer to the Send Thread. The Send Thread withdraws the packet from the Bounded Buffer and sends it to the server.

After the server processes the changes detailed in the Server section, it sends a TCP message and data using UDP to any active additional clients. Parsing the message passed from the server, the Client creates a “Client Receiver” object and begins performing the received operations using the Client Controller. If the message was for a delete, it calls a synchronized method to delete the data from the local directory. If it was for a download, a synchronized method in the Client Controller is called to begin the familiar process of creating Received Thread based on information contained in the TCP message.

The Receive Threads, upon receiving the data, deposit it into a Bounded Buffer. The Bounded Buffer is passed into a “File Writer” thread object. The File Writer calls to synchronized methods in the File Controller. If it is for a deletion, the file is deleted from the local directory. If it is for a download, the file is written to the local directory. While writing to the local directory, the second check for the pause and resume functionality is added, preventing the client from writing until the process has been resumed.

## Communication

The processes contained within each individual program have been detailed above. However, the communication between the two is just as intricate. Both the client and server must be able to receive and send TCP and UDP communications in order to accomplish the concurrency discussed later. However, if too many TCP and UDP communications are sent at the

same time the possibility of the wrong part of the system intercepting the wrong packet increases. As such, several considerations were taken in order to lessen the possibility of this happening.

In initial iterations of the program, several individual messages were sent for each piece of information (type of request, file name, file size, number of packets, number of blocks, etc.). Since this is an expensive process, these individual messages were combined such that the client and server only send TCP messages at limited times. The message sent is in the form of "A/N/S/B/M/M1/M2/.../Mn/EM/P1/P2/.../Pb". A is the action being requested. N is the name of the file on which the action is being performed. S is the size of the file. B is the number of blocks for that file. M is the determination of whether or not the file is modified. M1, M2, ..., Mn are the modified indices (used for delta sync). EM is to denote the end of the modified indices. P1, P2, ..., Pb are the number of packets in each block.

The UDP packets sent from client to server and from server to client are embedded with a two byte identifier. Originally, the idea was to send all the packets in all the blocks in one continuous thread. However, an issue arose in that bytes are only capable of handling values from -128 to +128 before overflowing. Since we would only be using the positive values, this range decreased to 0 to 128. To account for this, one of the bytes was used to identify the index the packet should be assigned and the other was a scale which acted as a multiplier. This allows us to identify 128<sup>2</sup> packets without overflow.

However, this issue was negated by changes in the overall design. After reworking the system, threads were used to maximize computational efficiency. As a result, at any given time, the identifier would only be limited to 65 at maximum. The reason for this was that each block is 4 MB. Each packet (minus the 2 byte overhead) is 65505 bytes. Distributing the blocks into bytes results in an approximate 65 packets, rendering the scale unneeded. However, it was left in the code, despite being deprecated. With this said, these identifiers are stripped and any padding (where the size of data is not fully equal to the fully 65505 bytes of the packet) is removed once the packets are received.

Before a client sends its packets through UDP to the server, it must wait to receive a TCP message indicating that the server is actually ready to receive and process the UDP packets. To make this indication, a TCP message is sent to the client to indicate the server is ready. Initially, the TCP receive methods were called in two places. However, this resulted in one of the methods intercepting the wrong message. To fix this, a Splitter object was created which gets updated by the TCP Receive thread in the client. This will be important in the concurrency section.

Certain operations, like the delete operation are conducted primarily through TCP commands. The client sends a TCP command to the server which instructs it to run a delete transaction on the database. The server sends a TCP command to the additional clients to indicate to run a Java NIO method to delete the specified file.

## Concurrency

While the processes above may have seemed sequential as described, concurrency is very present in the system. There are a total of 11 thread classes interspersed through the system. In

the client where multiple events can be fired by the operating systems. Each event creates a new thread for each file. Additionally, the Event Watcher and TCP based Receive Thread both run in parallel at all times. In the server, the Server Receiver thread continues to run in parallel to receiving the data from the client. Once the data is received and written to the server's database, it continues to synchronize with the additional clients.

Furthermore, the server must be able to synchronize communications with multiple clients. It receives data from the first client and sends it to additional clients. While this is occurring, the client communicates with the first client to ensure that no new data is sent until the server finishes sending the last data sent.

Another area where concurrency plays a role is through the use of the Splitter object. During its startup, a client creates a TCP based Receive Thread object. This thread receives all incoming messages and passes it to the Splitter object. The Splitter object allows for synchronized communication between the Client Receiver receiving data from the server and the File Controller sending data to the server.

### Miscellaneous Design

Some additional system designs included the use of a few helper classes. In particular, a "File Data" class was used to organize the files being transmitted on both the client and server sides. A "TCP Manager" and a "UDP Manager" were used to control the methods and sockets used for communication between the clients and server.

The File Data class had variables pertaining to the data of the file, the name, and its size. Inside were methods that could break the data into 4MB blocks or packets. Consequently, it also has methods to recombine the data into a single byte array. also contained methods to accomplish delta sync.

### Functionality

In accordance with the project's guidelines, there were five core functionalities that were implemented in the system: file sync, delta sync, suspend and resume, sync status, and error handling. In this section, there will be discourse on the specific features of the system that accomplish each of the requested functionalities. Any failures in the system will be noted briefly in summary, with more details provided in the Known Bugs and Issues subsection of the Testing portion of the document.

### File Sync

The requirement for file sync was that the files be synced continuously; be able to support binary and text files, each up to 200 MB in size; and that multiple files can be synced simultaneously for multiple clients. The first two points of this, continuous syncing, and file specification, are accomplished through the use of the Watcher Service. Any time a user makes a change to a file in the synchronized directory, the client immediately begins processing the change, uploading it to the server. This includes both text and binary files and large files up to the requested size.

The third is handled by the system's threading classes. The Event Watcher detects changes to the system and creates an Event Processor thread for each detected change. The Event Processor then creates a File Reader thread for each file.

### Delta Sync

The delta sync requirement was for the minimization of data, sending only changed blocks to the server and additional clients. The delta sync was implemented for the initial client to server communication, but due to time constraints was not implemented for the server to additional client synchronization. With that said, the implementation for the client to server communication is as follows.

First, upon start up, the client program adds any files in the directory to a HashMap data structure. This HashMap acts as a list of all files existing and is updated each time an operation is performed. Before it is updated, however, whenever the system detects a modification event, the current data in the HashMap is compared against the new data. There are three potential cases identified when comparing the data.

If the new data is the same size as the current data, it loops through each block to find any differences. If the new data is larger than the current data, it loops through the current data to find any differences and then adds the new blocks to the list of differences. Lastly, if the new data is smaller than the current data, it loops through the current block up to the removed blocks. The removed blocks are marked as negative numbers to indicate to the server that they should be removed from the data in the data table. This information is passed in the TCP message the client sends to the server (see the Communication Section in Design). The server then updates the data in the database accordingly.

### Suspend and Resume

The third requirement was for the implementation of suspend and resume functionality. This was implemented in the form of button controls in the client UI. This works by using a Synchronizer object, which is passed to the sections of code that read files from and write files to a client's local directory. This synchronizer object has a boolean which determines if the process has been paused and two methods to control this. One method, `checkIfPaused()`, is added so that during the read and write process the code checks if the boolean is true. If so, the threads running the read and write operation wait. The resume button notifies the threads to continue.

In the read process, where the data is read from a file into memory, the pause makes threads wait just before sending the data to the server through the Send Threads. In the write process, this is done just before the file is written to the local directory. As a result of this, the visual impact in the UI is more apparent in the read pause (pause initiated sending to the server) then it is the write pause (pause initiated when receiving data from the server). This is due to the write pause not occurring until all the data is received. The UI displays data being passed and since the pause cannot stop the server the send-receive communication from server to client continues even when the pause button is selected. The pause is still occurring, preventing the client from writing the file to the local directory.



The ability to pause and resume a delete operation also exists through the same methods. However, because of the simplicity of the operation this happens so rapidly that it is difficult for the user to respond in time. This was added though to ensure the functionality for the requirement was in place.

## Sync Status

This requirement requests that the system monitor and report the sync status of files being synchronized. Information such as progress, if the file is in or out of sync, if the file is in transit, and a sync timestamp must be provided. The event watcher monitors the sync status inherently by detecting any differences that arise. Whenever a file is transmitted a display has been added to the UI to show that the sync is in progress. Once complete, it shows that the directory is in sync with the server. The log in the UI shows the timestamp of when each synchronization process completes.

## Error Handling

Error handling was requested primarily for purposes of ensuring data consistency. Try-Catch blocks are used in general to assuage any failures that result in network outages or potential user errors. However, the general structure of the system ensures data consistency at all times. The way the system is set up to work is that the data is not written to a critical section until it has been received entirely. If the network crashes mid-process the no write operations are performed, meaning the data on the server is safeguarded.

## User Guide

The following section is a step-by-step walk through on how to compile, run, and operate the program.

### 1) Compile

In order to run the program, it is recommended that the user first compile the package. To do this, one needs to open a command prompt, change to the directory containing the cloudstorage directory. In the directory containing the cloudstorage directory, run the following command.

```
javac -d . CloudStorage.java
```

The user should then compile each subdirectory inside the cloudstorage directory in the following order.

```
javac cloudstorage\enums\*.java
javac cloudstorage\data\*.java
javac cloudstorage\network\*.java
javac cloudstorage\control\*.java
javac cloudstorage\server\*.java
javac cloudstorage\client\*.java
javac cloudstorage\views\*.java
```

### 2) Run the Server

Once this package is compiled, the user should run the following command to start the server first.

#### Windows

```
java -cp ";sqlite-jdbc-3.36.0.3.jar" cloudstorage.server.Server
```

#### Unix-based (Mac/Linux)

```
java -cp ":sqlite-jdbc-3.36.0.3.jar" cloudstorage.server.Server
```

If the user wants to reset the data table used to store the files in the server, they may run the same command with the argument “new” appended at the end. This is shown in the command below.

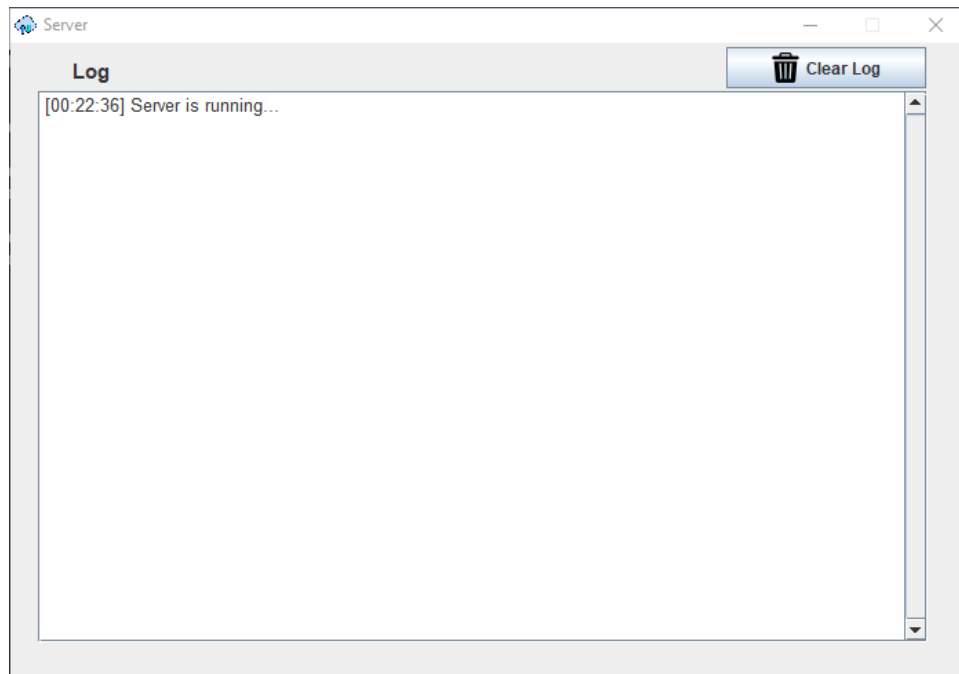
#### Windows

```
java -cp ";sqlite-jdbc-3.36.0.3.jar" cloudstorage.server.Server new
```

#### Unix-based (Mac/Linux)

```
java -cp ":sqlite-jdbc-3.36.0.3.jar" cloudstorage.server.Server new
```

The user will see the following window.



### 3) Start the client

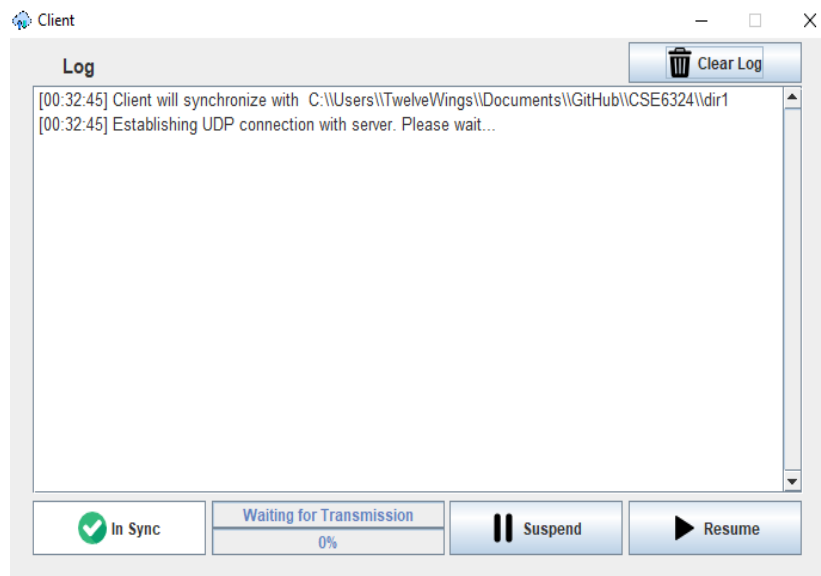
The user can then start the client program using the following command.

```
java cloudstorage.client.Client
```

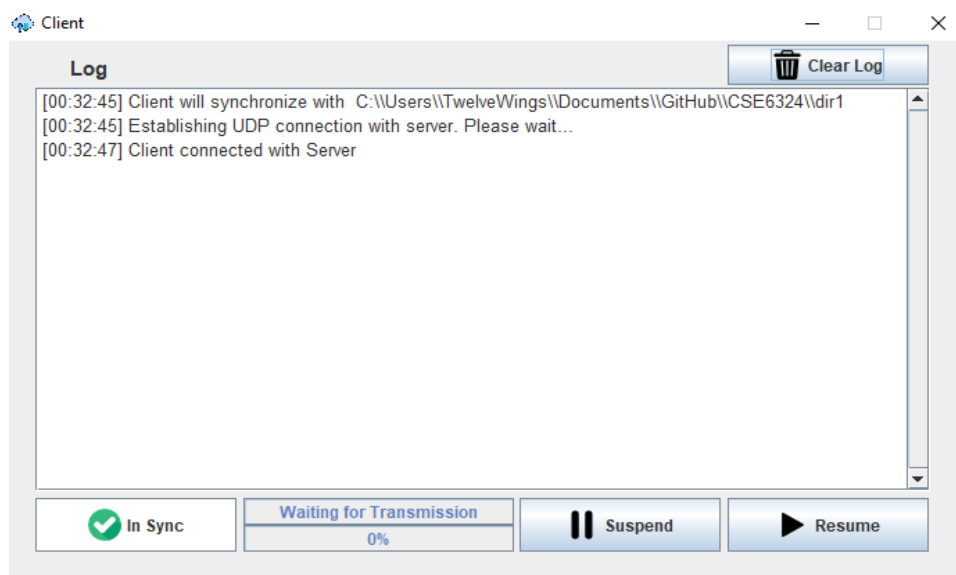
Once the program starts, the user will see the following windows.

#### 4) Select a directory

In the window, the user should select a directory and click “open”. Clicking cancel will close the client program. Once the directory is selected, the client will establish a TCP and UDP connection with the server. The user should wait until this finishes. It is also recommended that until this completes, no other clients are opened.



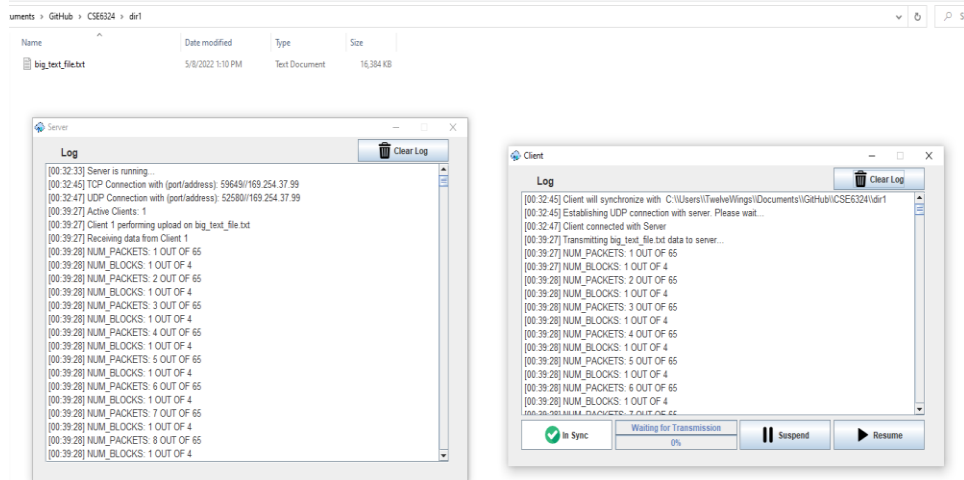
*Figure 1 Wait until finished.*



*Figure 2 Client is ready.*

5) Add file to selected directory.

The program will automatically start after a brief second to process to synchronization will begin.



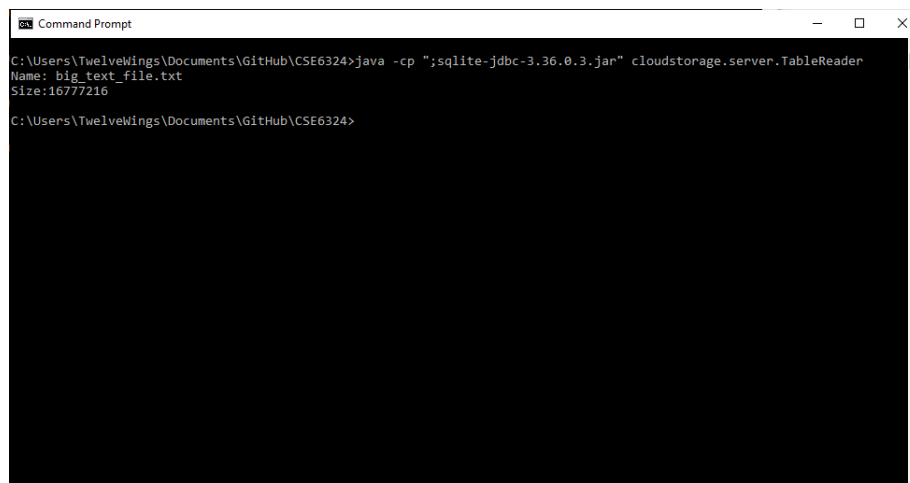
6) To verify that data has been added to the database on the server correctly, a helper program was created called TableReader. Table Reader effectively just runs a select query on the table and lists the results. To run it, the user enters the following command.

### Windows

```
java -classpath ";sqlite-jdbc-3.36.0.3.jar" cloudstorage.server.TableReader
```

### Unix-based (Mac/Linux)

```
java -classpath ":sqlite-jdbc-3.36.0.3.jar" cloudstorage.server.TableReader
```



## Testing

### Known Bugs and Issues

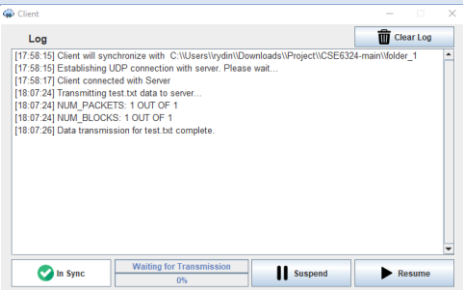
1. Delta Sync only occurs on client to server communication.
2. If more than two processes are simultaneously (in the form of a drag and drop in the OS) by the client at a given time, it may crash.
3. If a user makes changes too quickly, the client program will block the upload.
4. When modifying a text file, the indicated size may not change in the database but the data is correct.
5. The pause and resume functionality in the server to client synchronization may not work in all cases.

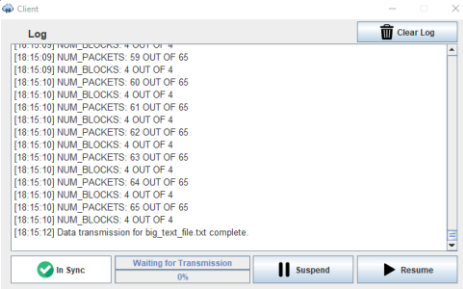
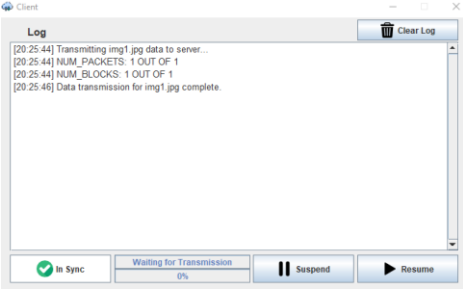
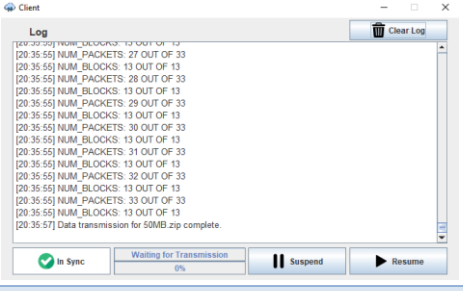
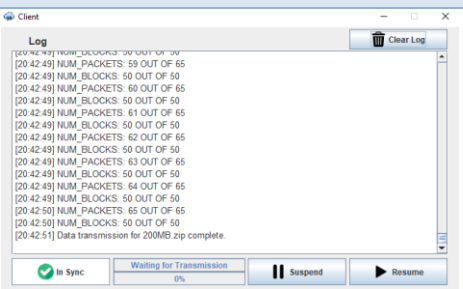
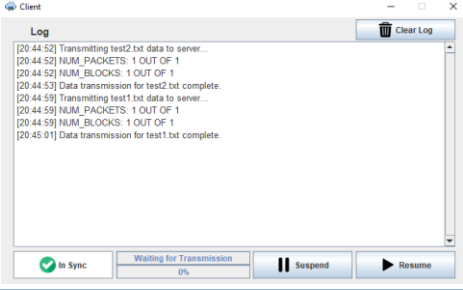
### Operating System Differences

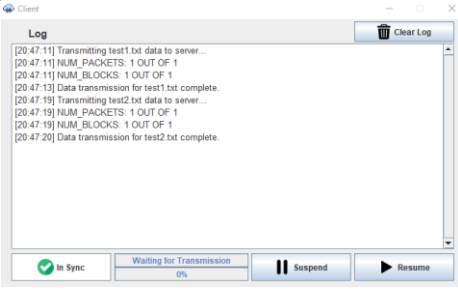
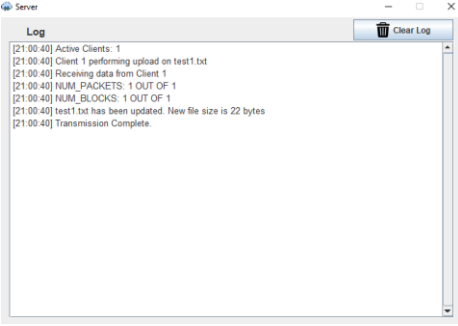
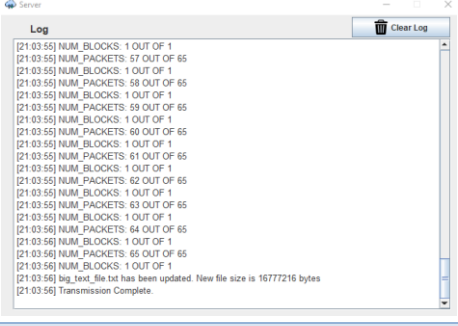
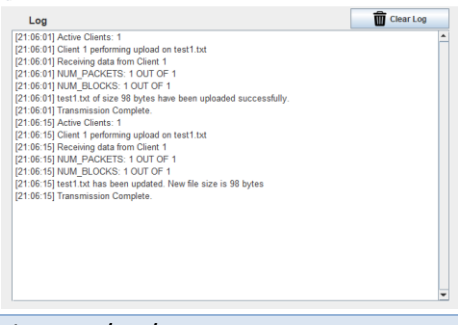
This system was developed primarily in Windows 10-11. Since the Java NIO API uses events generated by the operating system, it is possible that the system could perform differently on different OS. Since Windows was the primary development platform, all the test cases below were based on that. Due to platform availability issues, limited testing was performed outside of the versions of Windows noted. Brief testing in Linux was performed simple upload, modify, and delete operations. No testing was made on MacOS. However, since Mac, like Linux, is UNIX based, it is expected that the base functionalities work as intended there as well.

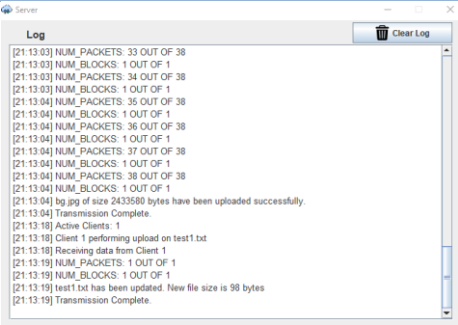
### Testing Cases

1. Upload/Modify

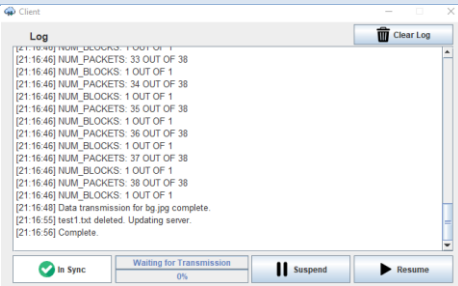
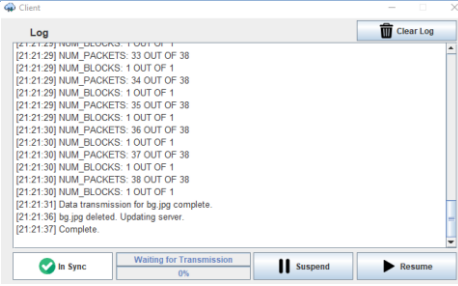
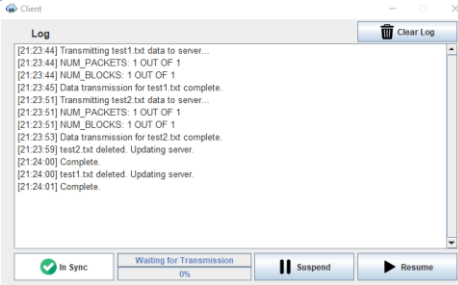
Use Case #	Expected Result	Observed Result	Screenshots
<b>1</b>	<b>Upload small .txt file ( &lt; 1 MB) – Tested on 04/09/2022</b>		
	A single .txt file less than 1 MB will be written to DB maintaining data integrity during the process.	As expected, the test.txt file has been written to the DB successfully in a single block.	 The screenshot shows a window titled 'Client' with a 'Log' tab. The log contains the following entries: [17:58:15] Client will synchronize with C:\Users\rydin\Downloads\Project\ICSE6324-main\folder_1; [17:58:15] Establishing UDP connection with server. Please wait...; [17:58:17] Client connected with Server; [18:07:24] Transmitting test.txt data to server...; [18:07:24] NUM_PACKETS: 1 OUT OF 1; [18:07:24] NUM_BLOCKS: 1 OUT OF 1; [18:07:26] Data transmission for test.txt complete. At the bottom, there is a status bar with a green checkmark and 'In Sync', a progress bar labeled 'Waiting for Transmission' at 0%, and buttons for 'Suspend' and 'Resume'.
<b>2</b>	<b>Upload large .txt file ( &gt; 1 MB) – Tested on 04/09/2022</b>		

	A single .txt file greater than 1 MB will be written to DB maintaining data integrity during the process.	As expected, the big_text_file.txt file has been written to the DB successfully in 4 blocks.	
<b>3</b>	<b>Upload small binary file ( &lt; 1 KB) – Tested on 04/09/2022</b>		
	A single binary file less than 1 KB will be written to DB maintaining data integrity during the process.	As expected, the img1.jpg has been written to the DB successfully in a single block.	
<b>4</b>	<b>Upload medium binary file ( &lt; 100 MB) – Tested on 04/09/2022</b>		
	A single binary file less than 100 MB will be written to DB maintaining data integrity during the process.	As expected, the 50MB.zip file has been written to the DB successfully in 13 blocks.	
<b>5</b>	<b>Upload large binary file ( ~ 200 MB) – Tested on 04/09/2022</b>		
	A single binary file equivalent to 200 MB will be written to DB maintaining data integrity during the process.	As expected, the 200MB.zip file has been written to the DB successfully in 50 blocks.	
<b>6</b>	<b>Upload two files simultaneously – Tested on 04/09/2022</b>		
	Two .txt files less than 1 MB will be written to DB simultaneously maintaining data integrity during the process.	As expected, the files test1.txt and test2.txt has been written to the DB successfully when uploaded simultaneously.	
<b>7</b>	<b>Upload two files in consecutive order – Tested on 04/09/2022</b>		

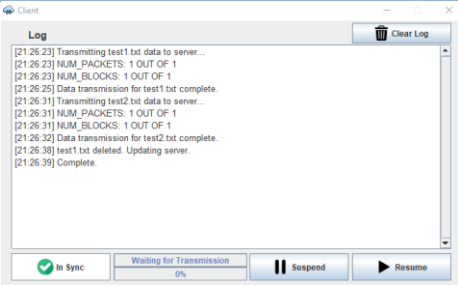
	Two .txt files less than 1 MB will be written to DB one after the other maintaining data integrity during the process.	As expected, the files test1.txt and test2.txt has been written to the DB successfully when uploaded in consecutive order.	 <p>Client Log window showing the following log entries:</p> <pre> [20:47:11] Transmitting test1.txt data to server... [20:47:11] NUM_PACKETS: 1 OUT OF 1 [20:47:13] Data transmission for test1.txt complete. [20:47:19] Transmitting test2.txt data to server... [20:47:19] NUM_PACKETS: 1 OUT OF 1 [20:47:19] NUM_BLOCKS: 1 OUT OF 1 [20:47:20] Data transmission for test2.txt complete. </pre> <p>Buttons at the bottom: In Sync (checked), Waiting for Transmission 0%, Suspend, Resume.</p>
<b>8</b>	<b>Modify small .txt file – Tested on 04/09/2022</b>		
	A single .txt file less than 1 MB which is already written to DB is modified. So, the modified block will be written to the DB.	As expected, when test1.txt file is modified, the server detected the modification and uploaded the modified block successfully.	 <p>Server Log window showing the following log entries:</p> <pre> [21:00:40] Active Clients: 1 [21:00:40] Client 1 performing upload on test1.txt [21:00:40] Receiving data from Client 1 [21:00:40] NUM_PACKETS: 1 OUT OF 1 [21:00:40] NUM_BLOCKS: 1 OUT OF 1 [21:00:40] test1.txt has been updated. New file size is 22 bytes [21:00:40] Transmission Complete. </pre>
<b>9</b>	<b>Modify large .txt file – Tested on 04/09/2022</b>		
	A single .txt file greater than 1 MB which is already written to DB is modified. So, the modified block will be written to the DB.	As expected, when big_text_file.txt file is modified, the server detected the modification and uploaded the modified block only 1 out of 4 (Delta Sync) successfully.	 <p>Server Log window showing the following log entries:</p> <pre> [21:03:55] NUM_BLOCKS: 1 OUT OF 1 [21:03:55] NUM_PACKETS: 57 OUT OF 65 [21:03:55] NUM_BLOCKS: 1 OUT OF 1 [21:03:55] NUM_PACKETS: 58 OUT OF 65 [21:03:55] NUM_BLOCKS: 1 OUT OF 1 [21:03:55] NUM_PACKETS: 59 OUT OF 65 [21:03:55] NUM_BLOCKS: 1 OUT OF 1 [21:03:55] NUM_PACKETS: 60 OUT OF 65 [21:03:55] NUM_BLOCKS: 1 OUT OF 1 [21:03:55] NUM_PACKETS: 61 OUT OF 65 [21:03:55] NUM_BLOCKS: 1 OUT OF 1 [21:03:55] NUM_PACKETS: 62 OUT OF 65 [21:03:55] NUM_BLOCKS: 1 OUT OF 1 [21:03:55] NUM_PACKETS: 63 OUT OF 65 [21:03:55] NUM_BLOCKS: 1 OUT OF 1 [21:03:55] NUM_PACKETS: 64 OUT OF 65 [21:03:55] NUM_BLOCKS: 1 OUT OF 1 [21:03:55] NUM_PACKETS: 65 OUT OF 65 [21:03:55] big_text_file.txt has been updated. New file size is 16777216 bytes [21:03:55] Transmission Complete. </pre>
<b>10</b>	<b>Upload a text file then modify it – Tested on 04/09/2022</b>		
	A single .txt file less than 1 MB will be written to DB. The written file is modified so that the modified block will be written to the DB.	As expected, test1.txt file is uploaded first. Modification is made on that file. The change is detected and uploaded successfully. <b>[BUG – File size doesn't update]</b>	 <p>Server Log window showing the following log entries:</p> <pre> [21:06:01] Active Clients: 1 [21:06:01] Client 1 performing upload on test1.txt [21:06:01] Receiving data from Client 1 [21:06:01] NUM_PACKETS: 1 OUT OF 1 [21:06:01] NUM_BLOCKS: 1 OUT OF 1 [21:06:01] test1.txt of size 98 bytes have been uploaded successfully [21:06:01] Transmission Complete. [21:06:15] Active Clients: 1 [21:06:15] Client 1 performing upload on test1.txt [21:06:15] Receiving data from Client 1 [21:06:15] NUM_PACKETS: 1 OUT OF 1 [21:06:15] NUM_BLOCKS: 1 OUT OF 1 [21:06:15] test1.txt has been updated. New file size is 98 bytes [21:06:15] Transmission Complete. </pre>
<b>11</b>	<b>Upload a binary file then modify a text file – Tested on 04/09/2022</b>		

	<p>A single binary file greater than 1 MB will be written to DB. Then, the .txt which is already present in the DB must be modified so that the modified blocks will be written to the DB.</p>	<p>As expected, bg.jpg file is uploaded first. Later, test1.txt which is already present is modified. The modification is detected and uploaded successfully.</p>	 <p>The screenshot shows a 'Server' window with a 'Log' tab. The log contains the following entries:</p> <pre>[21:13:03] NUM_PACKETS: 33 OUT OF 38 [21:13:03] NUM_BLOCKS: 1 OUT OF 1 [21:13:03] NUM_PACKETS: 34 OUT OF 38 [21:13:03] NUM_BLOCKS: 1 OUT OF 1 [21:13:04] NUM_PACKETS: 35 OUT OF 38 [21:13:04] NUM_BLOCKS: 1 OUT OF 1 [21:13:04] NUM_PACKETS: 36 OUT OF 38 [21:13:04] NUM_BLOCKS: 1 OUT OF 1 [21:13:04] NUM_PACKETS: 37 OUT OF 38 [21:13:04] NUM_BLOCKS: 1 OUT OF 1 [21:13:04] NUM_PACKETS: 38 OUT OF 38 [21:13:04] NUM_BLOCKS: 1 OUT OF 1 [21:13:04] bg.jpg of size 2433580 bytes have been uploaded successfully. [21:13:04] Transmission Complete. [21:13:18] Active Clients: 1 [21:13:18] Client 1 performing upload on test1.txt [21:13:18] Receiving data from Client 1 [21:13:19] NUM_PACKETS: 1 OUT OF 1 [21:13:19] NUM_BLOCKS: 1 OUT OF 1 [21:13:19] test1.txt has been updated. New file size is 98 bytes [21:13:19] Transmission Complete.</pre>
--	--	---	--

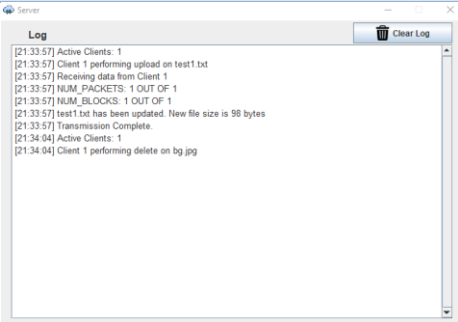
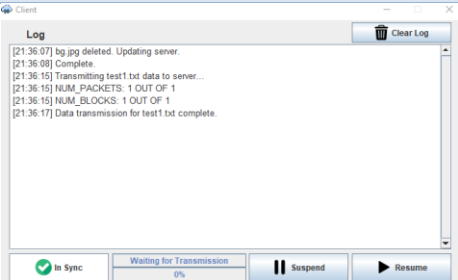
## 2. Delete a file

Use Case #	Expected Result	Observed Result	Screenshots
<b>1</b>	<b>Upload a file and delete a file – Tested on 04/16/2022</b>		
	<p>A single binary file greater than 1 MB will be written to DB. Then, the .txt file which is already present in the DB must be deleted when the local file is deleted.</p>	<p>As expected, bg.jpg file is uploaded first. Later, test1.txt file which is already present is deleted successfully.</p>	 <p>The screenshot shows a 'Client' window with a 'Log' tab. The log contains the following entries:</p> <pre>[21:16:45] NUM_PACKETS: 33 OUT OF 38 [21:16:46] NUM_BLOCKS: 1 OUT OF 1 [21:16:46] NUM_PACKETS: 34 OUT OF 38 [21:16:46] NUM_BLOCKS: 1 OUT OF 1 [21:16:46] NUM_PACKETS: 35 OUT OF 38 [21:16:46] NUM_BLOCKS: 1 OUT OF 1 [21:16:46] NUM_PACKETS: 36 OUT OF 38 [21:16:46] NUM_BLOCKS: 1 OUT OF 1 [21:16:46] NUM_PACKETS: 37 OUT OF 38 [21:16:46] NUM_BLOCKS: 1 OUT OF 1 [21:16:46] NUM_PACKETS: 38 OUT OF 38 [21:16:46] NUM_BLOCKS: 1 OUT OF 1 [21:16:48] Data transmission for bg.jpg complete. [21:16:55] test1.txt deleted. Updating server. [21:16:56] Complete.</pre>
<b>2</b>	<b>Upload a file and delete same file – Tested on 04/16/2022</b>		
	<p>A single binary file greater than 1 MB will be written to DB. Then, the written file must be deleted in local directory so that it gets deleted in the DB as well.</p>	<p>As expected, bg.jpg file is uploaded first. Later, this same file is deleted successfully.</p>	 <p>The screenshot shows a 'Client' window with a 'Log' tab. The log contains the following entries:</p> <pre>[21:21:28] NUM_PACKETS: 33 OUT OF 38 [21:21:29] NUM_BLOCKS: 1 OUT OF 1 [21:21:29] NUM_PACKETS: 34 OUT OF 38 [21:21:29] NUM_BLOCKS: 1 OUT OF 1 [21:21:29] NUM_PACKETS: 35 OUT OF 38 [21:21:29] NUM_BLOCKS: 1 OUT OF 1 [21:21:30] NUM_PACKETS: 36 OUT OF 38 [21:21:30] NUM_BLOCKS: 1 OUT OF 1 [21:21:30] NUM_PACKETS: 37 OUT OF 38 [21:21:30] NUM_BLOCKS: 1 OUT OF 1 [21:21:30] NUM_PACKETS: 38 OUT OF 38 [21:21:30] NUM_BLOCKS: 1 OUT OF 1 [21:21:31] Data transmission for bg.jpg complete. [21:21:36] bg.jpg deleted. Updating server. [21:21:37] Complete.</pre>
<b>3</b>	<b>Delete two files simultaneously – Tested on 04/16/2022</b>		
	<p>Two .txt files less than 1 MB will be written to DB. Then, both the written file must be deleted in local directory simultaneously so that it gets deleted in the DB as well.</p>	<p>As expected, test1.txt and test2.txt is uploaded to the DB. Later, both are deleted simultaneously.</p>	 <p>The screenshot shows a 'Client' window with a 'Log' tab. The log contains the following entries:</p> <pre>[21:23:44] Transmitting test1.txt data to server... [21:23:44] NUM_PACKETS: 1 OUT OF 1 [21:23:45] Data transmission for test1.txt complete. [21:23:51] Transmitting test2.txt data to server... [21:23:51] NUM_PACKETS: 1 OUT OF 1 [21:23:51] NUM_BLOCKS: 1 OUT OF 1 [21:23:53] Data transmission for test2.txt complete. [21:23:59] test2.txt deleted. Updating server. [21:24:00] Complete. [21:24:01] test1.txt deleted. Updating server. [21:24:01] Complete.</pre>
<b>4</b>	<b>Upload two files, delete one of the files – Tested on 04/16/2022</b>		



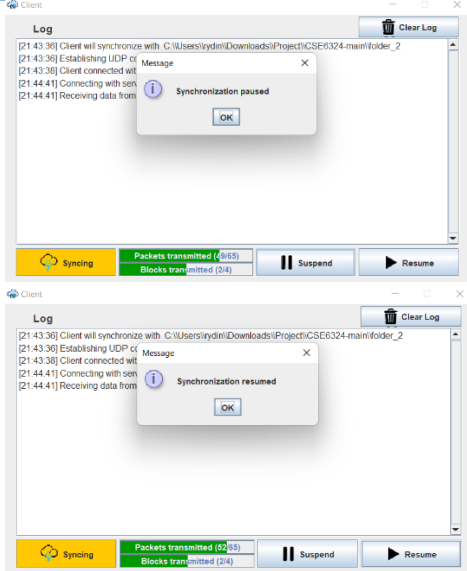
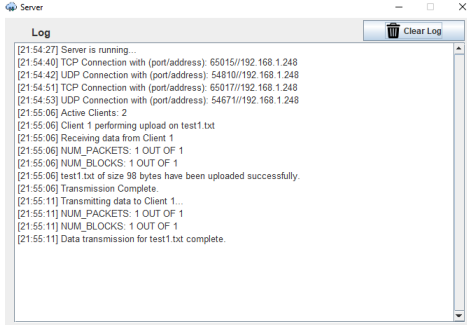
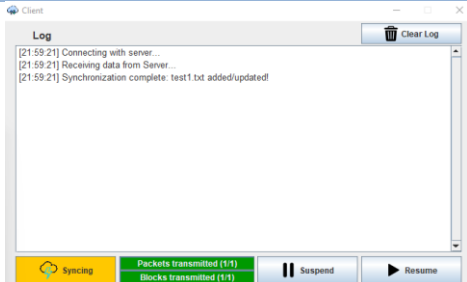
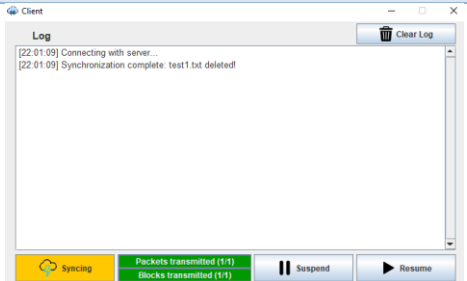
	Two .txt files are written to the DB consecutively. Then, one of the .txt file is deleted in the local directory so that it gets deleted in the DB as well.	As expected, test1.txt and test2.txt is uploaded to the DB. Later, test1.txt is deleted successfully.	 <p>The screenshot shows a 'Client' window with a 'Log' tab. The log contains the following entries: [21:26:23] Transmitting test1.txt data to server..., [21:26:23] NUM_PACKETS: 1 OUT OF 1, [21:26:25] Data transmission for test1.txt complete., [21:26:31] Transmitting test2.txt data to server..., [21:26:31] NUM_PACKETS: 1 OUT OF 1, [21:26:31] NUM_BLOCKS: 1 OUT OF 1, [21:26:32] Data transmission for test2.txt complete., [21:26:38] test1.txt deleted. Updating server., [21:26:39] Complete. At the bottom, there are buttons for 'In Sync' (checked), 'Waiting for Transmission 0%', 'Suspend', and 'Resume'.</p>
--	---	---	--

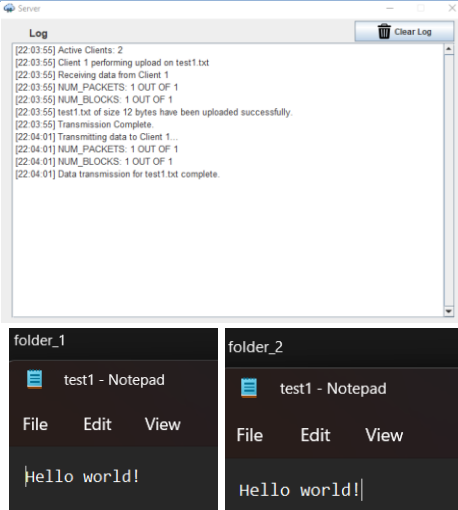
### 3. Mixed Case of the first two

Use Case #	Expected Result	Observed Result	Screenshots
<b>1</b>	<b>Modify the text file and delete the binary file – Tested on 04/16/2022</b>		
	A .txt file and binary file is present in the DB. Text file is modified, so that the modified block is written to the DB. After that is done, the binary file in the local directory is deleted, so that it gets deleted in the DB as well.	As expected, test1.txt file is modified first. Later, bg.jpg file is deleted successfully.	 <p>The screenshot shows a 'Server' window with a 'Log' tab. The log contains the following entries: [21:33:57] Active Clients: 1, [21:33:57] Client 1 performing upload on test1.txt, [21:33:57] Receiving data from Client 1, [21:33:57] NUM_PACKETS: 1 OUT OF 1, [21:33:57] NUM_BLOCKS: 1 OUT OF 1, [21:33:57] test1.txt has been updated. New file size is 98 bytes, [21:33:57] Transmission Complete., [21:34:04] Active Clients: 1, [21:34:04] Client 1 performing delete on bg.jpg.</p>
<b>2</b>	<b>Delete the binary file and modify the text file – Tested on 04/16/2022</b>		
	A .txt file and binary file is present in the DB. The binary file in the local directory is deleted, so that it gets deleted in the DB as well. After that is done, Text file is modified, so that the modified block is written to the DB.	As expected, bg.jpg file is deleted first. Later, test1.txt file is modified successfully.	 <p>The screenshot shows a 'Client' window with a 'Log' tab. The log contains the following entries: [21:36:07] bg.jpg deleted. Updating server., [21:36:08] Complete., [21:36:15] Transmitting test1.txt data to server..., [21:36:15] NUM_PACKETS: 1 OUT OF 1, [21:36:15] NUM_BLOCKS: 1 OUT OF 1, [21:36:17] Data transmission for test1.txt complete. At the bottom, there are buttons for 'In Sync' (checked), 'Waiting for Transmission 0%', 'Suspend', and 'Resume'.</p>

### 4. Synchronization

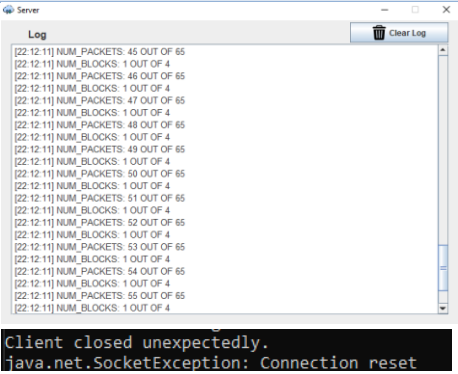
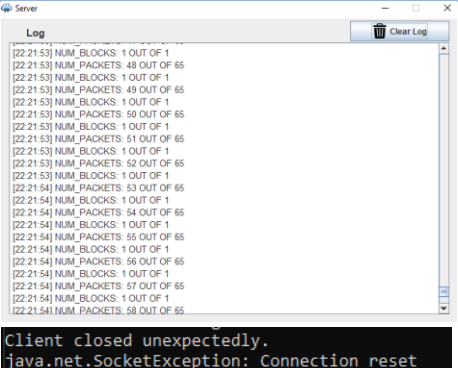
Use Case #	Expected Result	Observed Result	Screenshots
<b>1</b>	<b>Pause/Resume synchronization on active client – Tested on 04/16/2022</b>		

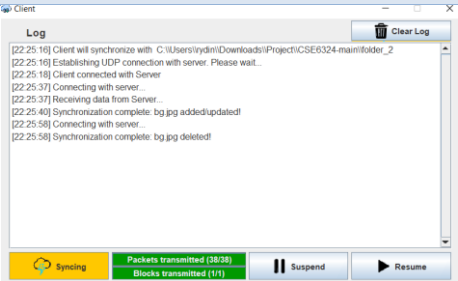
	<p>A .txt file greater than 1 MB will be written to DB. While it's being written, suspending it will put their upload thread to sleep. When resumed, upload thread is awakened, and the process should pick up at the same point where it was suspended. Data integrity will not be affected by pause/resume.</p>	<p>As expected, when uploading the big_text_file.txt suspend functionality stops the transmission and resume functionality resumes the transmission. <b>[BUG – Doesn't resume in client 2]</b></p>	
<b>2</b>	<b>Upload a file – Tested on 04/16/2022</b>		
	<p>Two clients are opened at a time and a .txt file less than 1 MB is written to the DB from client #1. After the file is written in the DB, the uploaded file is written in the client #2 directory from the server.</p>	<p>As expected, test1.txt file is uploaded in client #1. The server synchronizes the file with client #2 successfully.</p>	
<b>3</b>	<b>Modify a file – Tested on 04/16/2022</b>		
	<p>Two clients are opened at a time and a .txt file is modified in the client #1 local directory. After the modification is done, the modified block is written to the DB and sent to the client #2 from server.</p>	<p>As expected, test1.txt file is modified in client #1. The server synchronizes the modified file with client #2 successfully.</p>	
<b>4</b>	<b>Delete a file – Tested on 04/16/2022</b>		
	<p>Two clients are opened at a time and a .txt file is deleted in the client #1 local directory. The removed file will be deleted in the server followed by client #2.</p>	<p>As expected, test1.txt file is deleted in client #1. The server synchronizes and removes the deleted file from client #2 successfully.</p>	
<b>5</b>	<b>Check data on files after synchronizing with the server – Tested on 04/16/2022</b>		

	Two clients are opened at a time and a .txt file less than 1 MB is written to the DB from client #1. After the file is written in the DB, the uploaded file is written in the client #2 directory from the server. On opening that file in client #2, it must have the exact same data from client #1.	As expected, test1.txt file with data "Hello world!" is written in client #1 which synchronizes with client #2. Upon opening the files after synchronization, both contains the same data.	
--	--	--	--

## 5. Error Handling

### 5. a) Close client in middle of process

Use Case #	Expected Result	Observed Result	Screenshots
<b>1</b>	<b>Uploading a file – Tested on 05/08/2022</b>		
	When the client is closed during upload, the server stops the transmission as such and display error message.	As expected, the server stopped the transmission during big_text_file.txt upload and displays the error message successfully.	
<b>2</b>	<b>Modifying a file – Tested on 05/08/2022</b>		
	When the client is closed during modified upload, the server stops the transmission as such and display error message.	As expected, the server stopped the transmission during big_text_file.txt modified upload and displays the error message successfully.	

<b>3</b>	<b>Deleting a file – Tested on 05/08/2022</b>		
	When the client is closed during deletion, the server deletes the files as the delete operation happens much quickly.	The bg.jpg file is deleted on client #1 and closed immediately. However, due to the DB operation, the file in client #2 is deleted.	

## 5. b) Close server in middle of process

Use Case #	Expected Result	Observed Result	Screenshots
<b>1</b>	<b>Uploading a file – Tested on 05/08/2022</b>		
	When the server is closed during upload, the client stops the transmission as such and displays error message.	As expected, the client stopped the transmission during big_text_file.txt upload and displays the error message successfully.	The server was disconnected. Program terminated
<b>2</b>	<b>Modifying a file – Tested on 05/08/2022</b>		
	When the server is closed during modified upload, the client stops the transmission as such and displays error message.	As expected, the client stopped the transmission during big_text_file.txt modified upload and displays the error message successfully.	The server was disconnected. Program terminated
<b>3</b>	<b>Deleting a file – Tested on 05/08/2022</b>		
	When the server is closed during deletion, files aren't deleted and displays error message.	As expected, the connection between the client and server is lost and no files are deleted.	The server was disconnected. Program terminated

## Lessons Learned

## Future Works

As one would expect in any complex system, there are bugs and work needed for future development. Aside from fixing the bugs noted in the Testing Section, there are four additional features that will need to be implemented. First, the files in a synchronized directory should be deleted upon opening the client. This is to ensure that there aren't any initial synchronization issues. Next, the server should push files to the client as soon as the client opens. Third, a method

to request packets be resent if they are missing needs to be implemented when a program is receiving UDP packets.

Lastly, an update to the GUI did not fully merge in the code repository. Due to time constraints, this was considered non-essential as the requisite monitoring is still shown. The information is shown in different forms depending on if the client is sending or receiving.

## Conclusion

In this document, a discussion was provided on several aspects in regard to the cloud storage system. Insight was provided on the design as it pertained to the server program, the client program, the communication between programs, the system's concurrency, and miscellaneous details. This was mapped to the specified core functionalities requested for the assignment. Next, a user's guide was written, explaining how to operate the system. Then the test cases, including known bugs and issues, were detailed to ensure the functionality requested was operating as expected. Lastly, the authors spoke on the lessons learned and their intended future works as it relates to the system.