Trusted Bridge

Hao Zhuang, Erh-Li Shen, and Jin Wang

***Abstract*—Nowadays, there are many online file storage services providing convienent, accessible and giant *clouds* for people's daily usage. However, the convenience always comes with trust issues. There are many *hackers* around the world, sometimes your precious even private information is under peeking. This kind of hackers can also be extended to business services, e.g. data mining, machine learning, analysis for advertisment, or internal company employees misconducts.**

In this paper, we propose a trusted interface, so-called *Trusted Bridge*, which is built between user and different clouds (backend storage servers, Dropbox). This middle firmware creates a safe interface for storing and retrieving files. As a bridge, it does not store file directly, but guides the files with versatile encrypt methods to commute different clouds (or user's different accounts in the same cloud) automatically. More than a bridge, it is trustable because during this operations, the bridge partitions file and use hash function to determine the destination and replication strategy. The destinations are user's existed account within different cloud storage servers. When users want to fetch their files, the bridge know how to provide the tunnel with the key assigned. Because this procedures, those untrusted third-party storage systems only have (parts of) pieces of certain unordered files. So people can use the such untrusted services without worrying their privacy.

***Index Terms*—Untrusted Cloud File Service, Trusted Distributed File System-Oriented Design, Utilization Integration of Cloud Storages**

I. Introduction

NOWADYS, there are plenty of cloud storage service providers on the Internet, such as Dropbox, Google Drive, Skydrive or Amazon cloud storage. Most of them have easy-to-use user interface and support most of desktop and mobile platforms. However, under the mask of accessibility and reliability, we never know whether these service providers will analyze your contents or not even though they promised not to do so in the term of service. Take Google Drive for example, it indexes every file uploaded to its servers to optimize the searching efficiency. Even for those image files they could do OCR to filter out the important text content inside, and that is how Evernote works for the image in notes.

One of the most popular method in data mining is using term frequencyinverse document frequency product (TF-IDF)[1], which reflects the importance of a word in a document. By computing TF-IDFs for each file, the system could easily do data anaylsis for all the files that belong to specific users and learn what they like, what they working at, or anything personal.

We need a service that could take advantage from the convenience of cloud storage without loosing our privacy. Our approach is striping each single file into several fragments using a hashing algorithm with user-specified key and store those striped files in different cloud storage. Scattering the file could dramatically destroy the reliability of TF-IDFs since the weight in each fragment file could not efficiently reflect the whole content. What the cloud servers could see are encrypted file names and re-ordered file fragments.

II. Framework

The overall structure contains (1) client, (2) server, (3) backend storages. The structure is shown in Fig. 1

1) Client: The system (distributed working servers) provide users with uploading, downloading, deleting, newing folders, moving data opeations. Especially for the uploading operation usage, the user is required to set the "black-box" key and specified the security level normal, high, premium (higher level means more fragments a file will be partitioned, and it would lead to more time to process), which can be a figure or any combination of string and digits.Besides, the user need give the system access privilege to the external cloud storage servers (Google Drive, SkyDrive, Dropbox, etc). System will automatically maintain a table to record all the files' names and indices of their corresponding partitioned fragments.

2) Server: Server need be in charge of three responsibilies: determining how to partition the file based on the key given by the user and carrying out the work, communicating with clients and external cloud stroage by transferring kinds of data, assembling the fragments to the original file. In the server, it needs update a dictionary to store which storage server the

Hao Zhuang, Erh-Li Shen and Jin Wang are with the Department of Computer Science and Engineering, University of California, San Diego

Hao Zhuang (email: hazhuang@ucsd.edu). Erh-Li Shen (email: eshen@ucsd.edu) Jin Wang(email: jiw112@ucsd.edu)

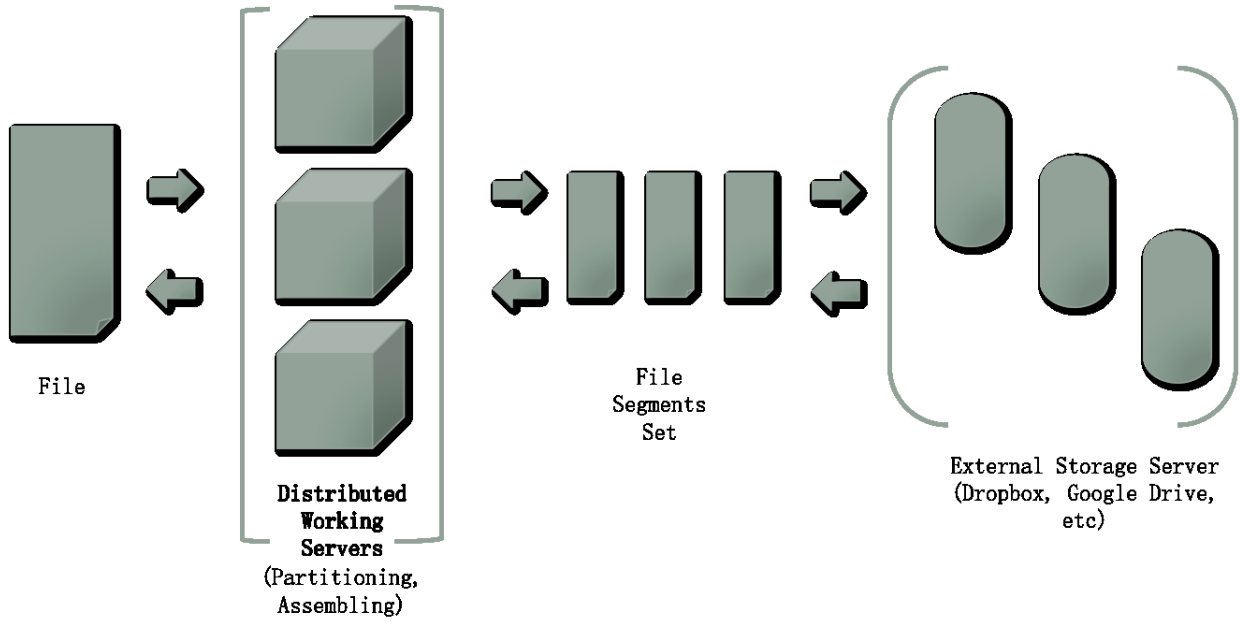

Fig. 1: Framework

fragment has been stored and maintains the encrypted version of users' table. The system still need perform a "heart-beat" function to update this encrypted table by communicating with users periodically.

3) External distributed server: External distrbuted storage servers provide interface to working servers. They are responsible for storing, replicating, backup files.

III. Software Artifact

We build a system contains as follows,

(1) A web interface for user to upload, download files and create folders, etc, like the functionality Dropbox website provides. When initializing his/her account, the user can choose the security level and integrate with his/her third-party cloud storage platforms.

(2) A server-side distributed files system to handle the file partitioning and assembling, using API from Dropbox to utilize their services. When user uploads a file, our server help partition (or encrypt if the security level is premium) this file, and perform as a *bridge* between the user and third-party cloud storage server. When user want to fetch a file, it invokes our server-side program to fetch the files fragment from different storage hosts and assemble them into the original file. The methodology is given in II. The blue parts of Fig. 2 and Fig 3 are built by us.

IV. Implementmentation

A. System Architecture

Trusted Bridge provides users a convenient online storage environment without losing their privacy. To achieve this goal, we implement the system with the following structure, please see Figure 2. We build all the blocks with blue color and take advantage from lab3 key-value storage system. The Trusted Bridge server and key-value storage server are built in sysnet clusters. The front-end web server is portable. It could be served right into sysnet clusters but it also could be run as a remote service on any Apache and PHP environment. The following sections will describe the detail implementation of each components.

B. Front-end Web Server

Trusted Bridge has an elegant web user interface to provide the user with an intuitive click-and-go file upload and download accessibility. The front-end web service is composed of three components: HTML/CSS for the interface template, JavaScript/jQuery for UI logic control, and PHP for handling file upload and download from client browsers. Figure 3 shows the architecture of the front-end web server.

We use Twitter Bootstrap to rapidly build the web user interface. Bootstrap contains HTML and CSS-based design templates for typography, forms, buttons, navigation and other interface components, as well as some useful JavaScript extensions. The HTML/CSS contents will be automatically updated to reflect the current system status

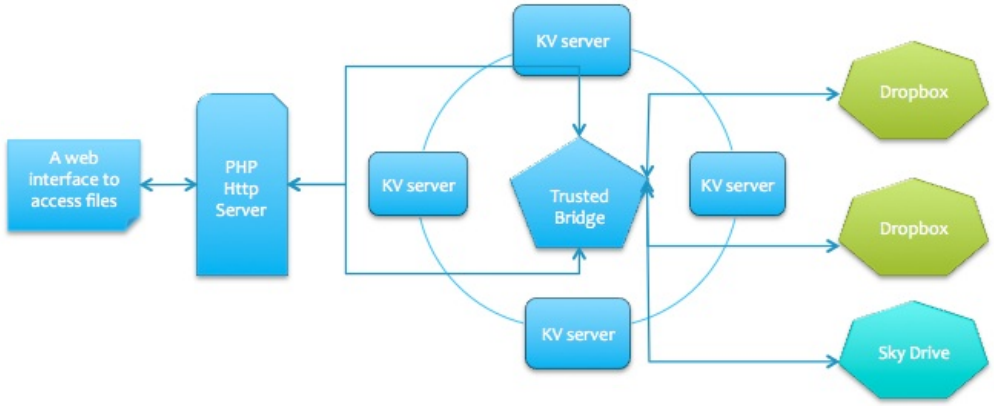

Fig. 2: Trusted Bridge architecture

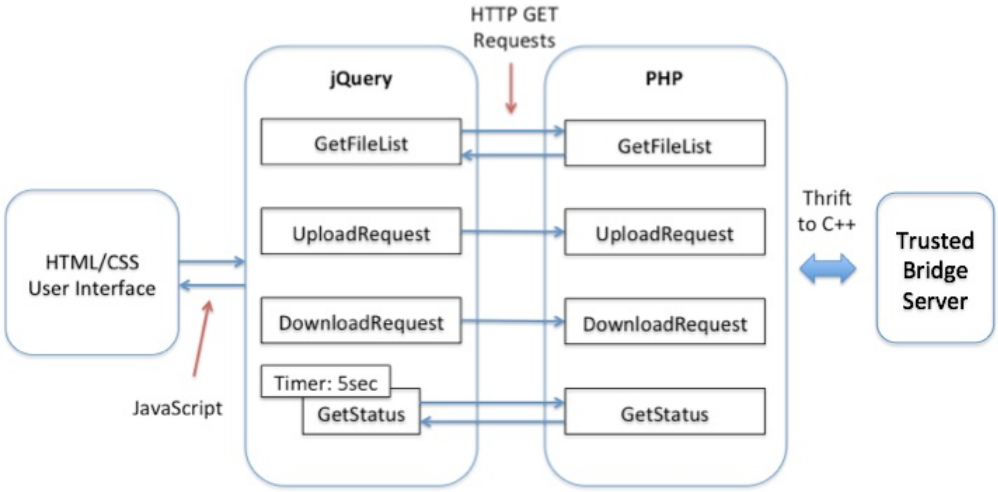

Fig. 3: Front-end web server architecture.

including the error message from Trusted Bridge server, the change of file list, upload and download progress, and etc.

With the help of jQuery, we could easily modify the HTML/CSS DOM objects dynamically. The UI event will trigger callback functions written in jQuery and do the related action. All the requests will be forwarded to a PHP program, which could arbitrarily access files on web server with appropriate permissions. The PHP program will talk to Trusted Bridge server through Apache Thrift protocol.

For example, when the user click on ”Add File” button, a file selector will pop out. After the file selection by the user, the file upload event in jQuery will be triggered and send the selected file from user's local storage to web server temporary folder. The PHP program will handle the upload request and send the file location to Trusted Bridge server. After the file partitioning and uploading to external cloud storage, Trusted Bridge server will response a OK message. The PHP program will just pass this response to front-end JavaScript with JSON format. When the front-end get the response, it will call GetFileList to get the latest file list in the system.

For downloading files, when the user click on the download button right next to the file name, jQuery will send a HTTP request to PHP server to ask for download action. The PHP program then forward the request to Trusted Bridge server through Thrift. After the file assembling task is finished, a downloadable intact will be saved in web server temporary folder. The front-end jQuery will check whether the file is ready or not every five seconds with the help of PHP server. Once the file is ready, the requested file will be automatically downloaded into user's local disk.

C. File Partition and Assembling

File Partitioning is the key point of this system and it inclues two main problems: first one is how to partition the file and the other one is how fine-grained the file need to be partitioned. For the first one, when a file

is uploaded to our server, we need to dispatch the file to different cloud storages. Our strategy is to read the file and truncate content on the fly, so that the large files do not occupy the large chunk of memory. For implementation, we also use JSON in Boost to control hierachical key value format, e.g. the order of segments for a file, the mapping from file to segments and different cloud services. First, the process get the parameter vector to specify how many segments for this file, and what the sizes are respectively. During the processs of spliting, it also invokes *GetSegmentName()* to get a encrypted name for such segement, which appear in the cloud storage. Others cannot judge the file by name. The method of *GetSegmentName()* is demonstrated in Sec. IV-D. The pseudo code is shown in Fig. 4.

For the merge of segments from different clouds, we fetch them from their repositories at first, Following the records from our key value storage, assemble them together accordingly.

D. Segment Naming, and Ordering

As each file can be partitioned into 32 segments or more, to make segment name unique comes to be a problem. Because we do not want any file to be over-written. The other question is to maintain the generating ordering of segments, which is used to assemble the file. Any small mistake here will results in failing in recovering the original file. For the first question, two conditions shall meet: first one is segment name shall be unique, sencond is that the name shall be random. SHA2-256 provide sufficient functionality that different input always gives out different hash value, which can satisfy our first condition. For the second one, we at the beginning adopt the method to give a random number among a pretty large range. After several experiments on random number generating, due to the bad distributuion of random number, some number will be generated the same in the second time. Then we change the strategy to take the system CPU counter as the input that always is unique as the seed of hash function. In this way, we first get the system CPU counter and get the corresponding unique hash value, later this will act as the name of segment.

Segment Ordering provides the mapping structure when assembling the file. This requires a stable data structure or some other kinds of format maintaing the segment generating order. Each time we get a new segment, a new item as talked above is enqueued to the uploading task queue with counter for this queue. As soon as the item successfully gets into it, a key$-$value pair $<$segment name, counter$>$ (segment with counter i means it is the i-th part of the file). Till finish partitioning file into segments, we collect all of these pairs as json array into the json object whose key is the name of the file. When trusted bridege server want resemble file, it is straigtforwatd to get the order of segment by parsing the json object by given the file name.

E. Key-Value strucutures

To provide more convenient service for trusted bridge server, we design the following key value pairs. The first kind $<$username clouds, token array for the cloud storage$>$ is used to record user's tokens for different cloud storage access. The second kind $<$username files , filename lists$>$ record the list of filenames the user has uploaded. The key for the third one is username plus filename, the value corresponding to it is the list of segment name. The fifth pair's key is username plus segment name, the value is the place of the cloud who has this segment. The last kind is the pair username and the key it specified when registraing a new user.

V. Evaluations

A. Measurement Method

To get more accurate performance measurement result, we design the experiment very carefully. First, we assume that Dropbox API will compare user data with what the already had in the cloud before the API actually upload the file. It is reasonable for them to do so to avoid massive redundant uploading bandwidth. However, for our measurement, this is not a good property because the upload speed will be much faster than it should be after the first upload action. To avoid this kind of situation, we generate a new random file with a random file name from system random source /dev/urandom each time before we upload it. Since each time both the file name and the content is different, the Dropbox API has no chance to compare the data before upload. Then we can measure the actual time consuming during upload/download progress.

Second, as the suggestion from Professor Alex C. Snoeren, we also scramble the order of uploading and downloading tasks. Sequential uploading or downloading files with the same file size could not ignore the fact that the network between sysnet clusters and Dropbox is not always stable. The file-transfer-speed statistics collected could be affected by the current network condition. We generate a random test order for different file size to avoid this problem. In the following evaluation, we run each test at least 100 times and get the average data from the results.

```
fp = fopen(file, "rb"):
for (int it = 0 ; it < chunk_size_array.size ; it++){
  buffer = malloc(chunk_size_array[it]);
  file_name = GetSegmentName(user_key)
  // treating every file as binary
  segment_file = fopen(file_name, "wb")
  fwrite(buffer,1,current , segment_file);
  fclose(segment_file);
  free(buffer);
  // use hierachical Key Value pair by JSON
  // It provide a file contains what segments,
  // their order, and position.
  // Simplify the process by register_to_key_value() as follows
  register_to_key_value(user, cloudID, file , segment_order , segment_name);
}
```

Fig. 4: Pesudo Code of File Spliting

```
key                                  Value
-----------------------------------------------------
${username}_clouds                 (JSON) clouds
${username}_files                  list<filename>
${username}_${filename}            list<segment_name>
${username}_${segment_name}        cloud_id
${username}_key                    any string
-----------------------------------------------------
```

Fig. 5: Key-Value Storage Format

```
 early_cloud  {
 "clouds":[
 {"id":0, "type":"dropbox", "token1":"12345",  "token2":"67890" },
 {"id":1, "type":"dropbox", "token1":"12345",  "token2":"67890"},
 {"id":2, "type":"google" , "token1":"12345",  "token2":"67890"}
 ]
 }
 early_files              list<"file1.avi","file2.txt","file3.jpg"...>
 early_file1.avi          list<"seg1.seg","seg2.seg","seg3.seg"...>
 early_file2.txt          list<"seg1.seg","seg2.seg","seg3.seg"...>
 early_seg1.seg           0
 early_seg2.seg           1
 early_seg3.seg           2
```

Fig. 6: Example of Our Key Value Pair

Third, we run the test with different Dropbox accounts and different sysnet cluster machines. Keep running a script that uploads and downloads small files continuously maybe will soon be noticed by Dropbox servers. To avoid the possible limitation on file-transfer-speed by the service providers, we switch the account randomly and test the script on different machines with different IP address.

B. Dropbox Overhead and Performance

We first found that the Dropbox Java API has a huge overhead on each uploading and downloading operation. This overhead dramatically slows down the performance of Trusted Bridge. We do not have enough time to implement the API by pure HTTP requests, so we decide to test the Dropbox API overhead and assume that we could minimize this overhead in the future with better implementation. The method to estimate the overhead is to generate a empty file by linux touch command. The file will be created with 0 Byte disk occupation

and online contains metadata information. The average upload time for one single upload task with one empty file is 3.45 seconds. The average download time for the same file just uploaded is 3.12 second.

To test the general performance of Dropbox uploading and downloading performance, we generate a series of random files with size 1Byte, 1KB, ad 1MB to 128MB. The result is shown in Figure 7. All the statistics are in Mbps unit. Averagely the download speed is a little faster than upload speed. The difference is about 3Mbps, which is not so significant. The interesting fact here is the performance is very bad when uploading or downloading small files. The speed is converged at about file size 4M to 8M. According to this phenomenon, we believe that Dropbox API will do file upload and download in chunk. The chunk size should be around 4M to 8M.

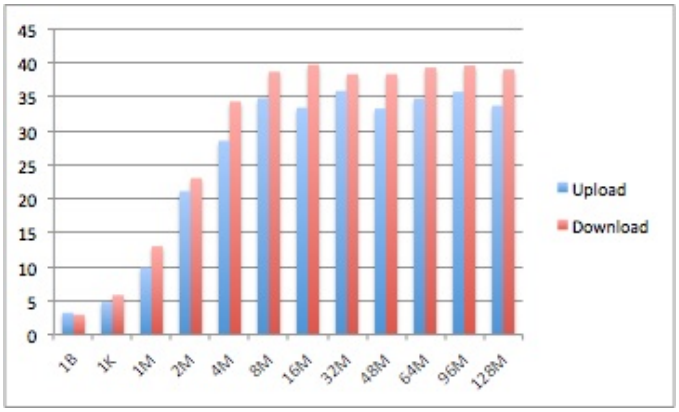

Fig. 7: Dropbox upload and download performance

C. Trusted Bridge Performance

Figure 8 shows the upload speed comparison between Dropbox and Trusted Bridge server. The performance of Trusted Bridge on uploading small files are significantly worse than using pure Dropbox API. Figure 9 shows the download speed comparison. Very similar pattern is showed in the graph as uploading test. The best performance lose is 9.57% with the file size 128M. The worst performance lose for uploading, however, is 97% with file size smaller than 4M. The best performance lose is 8.62% with the file size 128M. The worst performance lose for uploading, however, is 96% with file size smaller than 4M.

The main reason of the performance degradation is that we separate each uploaded files into 32 pieces and upload them through Dropbox API one by one. The Dropbox API overhead will dramatically affect on our system performance. We believe we could find a better way to handle external cloud storage, so here we assume that we could remove the effect of the Dropbox uploading and downloading overhead. Because each file needs 32 upload or download task, we compare the speed performance between Trusted Bridge and Dropbox uploading or downloading the same actual file size. For example, uploading 128M file to Trusted Bridge server compare to uploading 4M file to Dropbox. The result is shown in Figure 10 and Figure 11. In this comparison, we could found that Trusted Bridge server only lose averagely 8.2% on uploading and 9.7% on downloading speed. This result shows the overall partitioning and assembling performance do not degrade the system performance too much. With our robust file scrambling algorithm, the user could save their files in their own cloud storage without losing the privacy. What he or she scarifies is just about 8 to 10 percent performance lose, which could hardly be noticed especially when user run our service in background like how current Dropbox or Google Drive works.

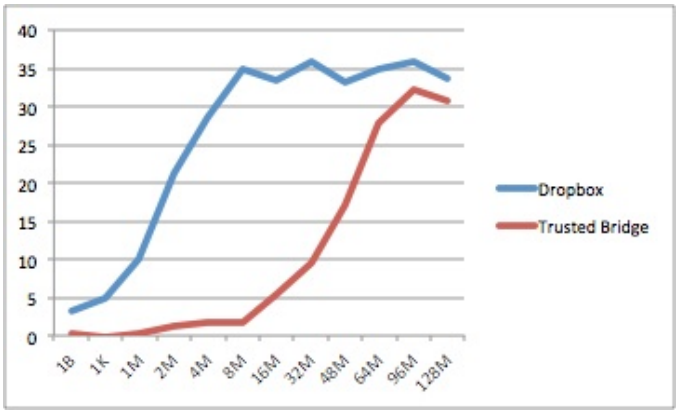

Fig. 8: Upload speed comparison

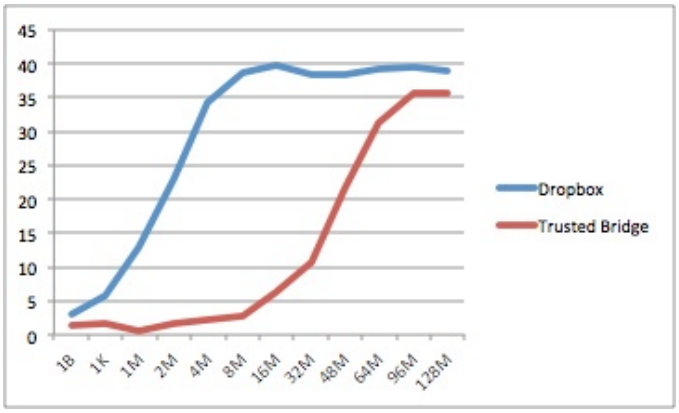

Fig. 9: Download speed comparison

VI. Discussion and Future Work

A. Upload Task Queue

As each file would be partitioned into 32 or more segments and then they shall be pushed to external kinds

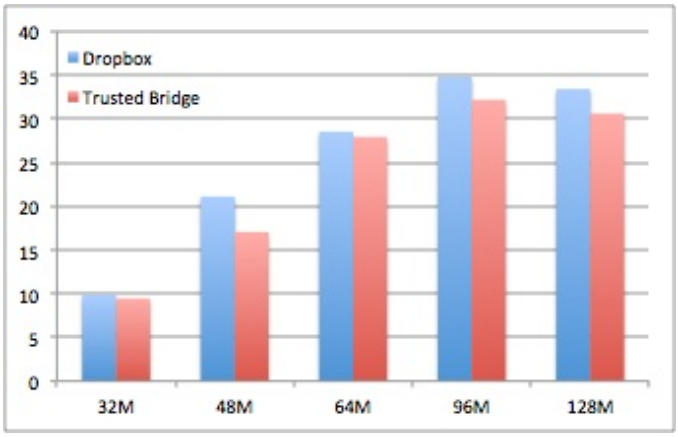

Fig. 10: Calibrated upload speed comparison. The file size here is the size uploaded to Trusted BRidge server

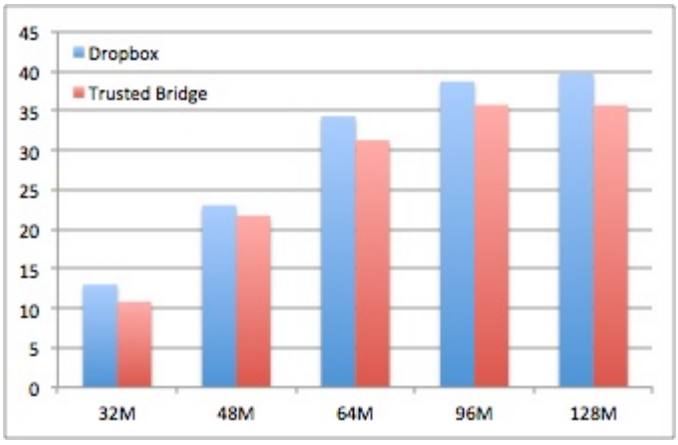

Fig. 11: Calibrated download speed comparison. The file size here is the size downloaded from Trusted BRidge server

of cloud storage servers, we take this way to loosely separate these two kinds of work. The main thread of trusted bridege server still take the responsibility for generating the segments. The reason why we just use one thread here is to decrease the possibility of erronously partitioning the file. Consider that several threads are working for generating new segments, shared lock for the file need be maintained for synchronizing reading the file. This will lead to a high overhead. What's worse is when a thread crashed, the segment it's responsible for will be abnorammly generated or some other kinds of errors may happen. If so, it will violates the most imporant part for trusted bridge server, what is to store complete information about the file and related segments. While, the independent segments generated by trusted bridege server can be severd with sveral threads. When initializing the trusted bridge server, there threads are created for dealing with the segment uploading task. A onewriterrmorereader task uploading queue will be created as the working pool for trusted bridge server. This queue maintains one mutex and one condition variiable for threads sharing. When going into "enque" opeartion, the thread need first acquire the mutex for this queue, if it's permitted, one new entry that contains the pair ¡username, segment name, cloud id to store current segment¿ is pushed back to the queue, meanwhile, ñotifytoneẗor the condition variable is called to wake the thread waiting for new segment. If this queue is full, resize opeartion will be invoked. For "deque" operation, the mutex still need be checked firstly. Simiarly, it gets the access if allowed and then check segment ready for being uploaded. If there exists one, the item records the uploading specification will be removed from the queue and be transfereed to uploadToServer function.

B. Fault-Tolerance Scheme

Two big issues here need be more focused. One is that we cannot lose anything about the mapping strucuture between file and related segments. We solve this by using a replicated key value distributed storage server. As one trusted bridge server just talks to one key value storage server one moment in time, the replicated storage server will help syncrhonize writing key value pair by communicating with each other in order to prevent some storage server crashing. This method will enormously get the risk out. The second one is to keep data consistent during the period of generating the segment. For example, server may fail in trasmitting segments to the cloud. To fix this, a thrad can be used to monitor and periodically check whether the segment has been put on the external cloud storage. If it has been there, the server begin to update the item to array in the mapping structure for this segment. If it is not, the trustd bridge server shall resend current current segment. When learning that all the segments for one file have been pushed to the cloud, the json object describing one file with corresponding segments will be put into the key value storage server. The user will not get the information that the file has been uploaded successfully till we get the commit acknowledgement from the key value storage server.

C. Text Mining

Text mining methods, there is no major increased computational complexity when analyzing many short documents, compared to the same number of words in fewer documents.A much more important issue is whether individual documents are well-defined in a meaningful way. Most text mining methods work best on documents whose length is a paragraph to a page.

Too short documents like tweets cannot be understood well without context. Too long documents like books lack homogeneity and need to be divided into segments. The more segments the file is divided, the more diffculty will put on the analyzers. Take the TF(term frequency)-IDF(Inverted Document Frequency) algorithm to weight the keywords for a file as an example. The partitioning will change the term frequency for keywords, which will definitely change each keyword's weight. Besides, as we distributed segments to different servers, the IDF value would still change. Because these two values are changing together, it will direct user the wrong way analyzing the file, which is the target trusted bridge server want serve.

D. Programming Language Integration

In this implementation, we totally use at least four different programming language including HTML/CSS/-JavaScript, PHP, C/C++, and Java. That was because we first write our Trusted Bridge server in C/C++ as an extension of lab3 key-value storage system. C/C++ is powerful and efficient, but it lacks cloud storage API support and HTTP request handler. The difference of programing environment slows down the development progress and the system could not take advantage from compiler level memory and CPU optimization. If we have the opportunity to redesign this system, we would like to rewrite Trusted Bridge server in Java, and the dynamic web server in JavaServer Pages (JSP). In this case, Trusted Bridge could be integrated with Dropbox or Google Drive API in JAVA much more closely. The front-end server also could talk to Java Servelet directly without the Thrift protocol overhead. The new system could be implemented with HTML/CSS/JavaScript, JSP, and Java, which is much more easy to maintain and much more efficient.

VII. Conclusion

In this project, we derive our own idea and build a trustful file system *Trusted Bridge*, which utlizes unstrustful popular cloud storages, e.g. Dropbox.The approach is to partition the files, encrypt, and then send them into different cloud systems. For a single file, we do not save it, and the cloud storages only contains a certain part of file segments. In this project, we use C/C++ to handle the backend design and use wrap interface to invoke Jave, Python for API from Dropbox, Google Drive. The PHP is used to handle the HTTP requests. Experiement shows that after removing the Dropbox API overhead, the user could save their files in their own cloud storage without losing the privacy. What he or she scarifies is just about 8% to 10% performance lose, which could hardly be noticed especially when user run our service in background like how current Dropbox or Google Drive works.

References

[1] Term FrequencyInverse Document Frequency (TF-IDF). http://en.wikipedia.org/wiki/Tf-idf.

VIII. Appendix

In the appendix, we provide a snapshot of the code to further utilize multi-thread environment and speedup backend for handling the file uploading and downloading. There are Locking Queue and corresponding Pthread programming, which are discussed in IV-D .

A. Locking Queue

The locking queue class for future usage.

```
template <typename Data>
class oncurrent_Queue{
private:
    queue<Data> _queue;
    mutable boost::mutex _mutex;
    boost::condition_variable _cvariable;
public:
    void enque(Data const& data){
 boost::mutex::scoped_lock lock(_mutex);
 _queue.push(data);
 lock.unlock();
 _cvariable.notify_one();
    }

    bool empty() const{
 boost::mutex::scoped_lock lock(_mutex);
 return _queue.empty();
    }

    bool try_deque(Data& dequed_value){
 boost::mutex::scoped_lock lock(_mutex);
 if (_queue.empty()){
     return false;
 }
 dequed_value = _queue.front();
 _queue.pop();
 return true;
    }

    void wait_and_deque(Data& dequed_value){
 boost::mutex::scoped_lock lock(_mutex);
 while(_queue.empty()) _cvariable.wait(lock)
    ;
 dequed_value = _queue.front();
 _queue.pop();
    }


};

void *checkUploadQueue(void* context){
```

```
    int id = *(int*)thread;
    while(true){
 if (!uploadQueue.empty()){
     string ret;
     uploadQueue.try_deque(ret);
 }
    }
}
```

B. *Pthread*

The pthread can be utilized to listen to the upload queue.

```
TrustedBridgeHandler t(storageServer,
  storageServerPort);
const int NUM_UPLOAD_THREADS = 3;
pthread_t threads[NUM_UPLOAD_THREADS];;
for (int i = 0; i < NUM_UPLOAD_THREADS; ++
  i){
    int rc = pthread_create(&threads[i],
  NULL, &TrustedBridgeHandler::
  checkUploadQueue, &t);
    if (rc){
 exit(-1);
    }
}
```