Designing a Dropbox-like File Storage Service

A Performance and Cost study

Hassaan Bukhari
School of Information Technology
York University
Toronto, Canada
hassaanb@yorku.ca

Fariborz Khanzadeh
School of Information Technology
York University
Toronto, Canada
fariborz@yorku.ca

Alejandro Ramirez
School of Information Technology
York University
Toronto, Canada
aramirez@yorku.ca

Abstract — In this paper we present a reference architecture of a Dropbox-like file storage system by identifying its core components and communication protocols. We also present a deployment architecture that explains how our Dropbox-like file storage system can be deployed in the Cloud. We chose Amazon as a Cloud Provider to carry out a feasible implementation. We have also evaluated key performance metrics by presenting hypothetical scenarios. Optimal assumptions and estimates of user traffic and data storage utilization are made in the paper. In addition, cost parameters of cloud resources are evaluated with the actual dollar-values, based on the pricing information available on Amazon Web Services.

Keywords — Dropbox, Cloud File Storage, Synchronization, Message Queuing, Amazon Web Services, Amazon S3, Amazon SQS, DynamoDB, Performance, Cost.

I. INTRODUCTION

Cloud file storage services have become very popular recently as they simplify the storage and exchange of our digital resources among us and our multiple devices. The shift from using single personal computers to using multiple devices with different platforms and operating systems such as smartphones and tablets, and their portable access from different geographical locations at any time is believed to be accountable for the massive popularity of cloud storage services. These services usually provide a complete set of tools for file storage, sharing, and automatic synchronization as their three key features. At the same time they take advantage of the benefits that cloud solutions provide inherently such as availability, redundancy, and scalability.

However, not much is known about the internals of commercial solutions such as Dropbox, Google Drive, OneDrive, etc. as they are proprietary and closed. These solutions rely on architecture and algorithms that are not visible to the outside world. Therefore, in order to study and propose a reference architecture for such services, we have researched the architecture of the open source alternatives such as ownCloud, SparkleShare, Syncany, and StackSync.

In terms of system deployment, we design a physical environment based on our reference architecture and some of the Amazon's first services that are offered at a large scale, namely Amazon S3, Amazon SQS, Amazon EC2 and Amazon DynamoDB. We explain how each of these services functions and is utilized by our solution.

Thereafter, we will analyze the performance of the application based on the specifications of computing and data storage services made available by Amazon on their website. We will develop estimates and will simulate scenarios to show performance variations. Based on those scenarios, we will calculate monthly cost of Cloud services using the advertised pricing information on Amazon website [18]. At last, we assess the profitability of the application at a high-level.

The remainder of this paper is organized as follows: Section II provides a summary of the related work. Section III presents our reference architecture. Section IV presents the proposed deployment architecture to build our system on top of Amazon Web Services. Section V presents the analysis of performance and cost metrics of the application with estimates and scenarios. Finally, the conclusion is provided in section VI.

II. RELATED WORK

Little is known about the design, architecture, and implementation of commercial personal cloud storage systems. Drago et al. [1] have studied the architecture of Dropbox by conducting performance measurements based on network traffic traces. According to them, Dropbox file synchronization is built using third-party libraries such as librsync [2]. However, not much is known about the details of the file synchronization mechanism and metadata organization. The same is true about other personal cloud storage providers such as Google Drive, OneDrive, etc. Nevertheless, most of these providers decouple the control flow from the data flow by storing the files on separate storage servers, and processing the data on separate application servers. For example, Dropbox employs Amazon S3 for their data storage, and uses its own private cloud for file synchronization and metadata management. Li et al. [3] has represented the high level architecture of the Dropbox as seen in Figure 1.

Dropbox uses a client application to monitor file changes in specified folders. After a change notification is received, the client application indexes the affected files. Then, the file is compressed and sent to Amazon S3, and the file metadata is sent to the Dropbox private cloud. The maximum size of an object in Dropbox is 4MB. Files larger than that are split into chunks of 4MB each. Dropbox reduces the amount of data exchange by transmitting the updated chunks of a file only. It also keeps a local copy of metadata information in each device. Dropbox uses a cluster of notification servers for pushing notifications about the updates to the clients. Lopez et al. [4],

on the other hand, propose using a messaging middleware for change notifications as decoupling message delivery from message processing is a key requirement for scalability. Using a messaging middleware simplifies the overall architecture since it also provides load balancing.

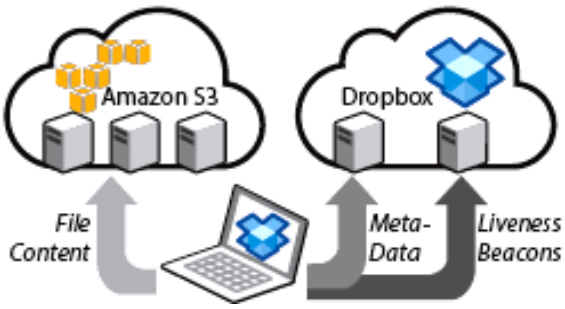

Figure 1. Dropbox High Level Architecture

III. LOGCAL ARCHITECTURE

Our reference architecture for a Dropbox-like cloud storage system consists of five main components: Desktop Client Application, Synchronization Service, Message Queuing Service, Metadata Database, and the Cloud Storage back-end. The proposed architecture of our system with the main components and their interaction is presented in Figure 2 below. The Client Application and the Synchronization Service interact through the Message Queuing Service. The Synchronization Service also interacts with the Metadata Database for data persistence. Client Application directly interacts with the Cloud Storage back-end to upload and download files. Our reference architecture is based on a loosely coupled service oriented design that enables us to implement and deploy it on different types of clouds including public, private, and hybrid clouds.

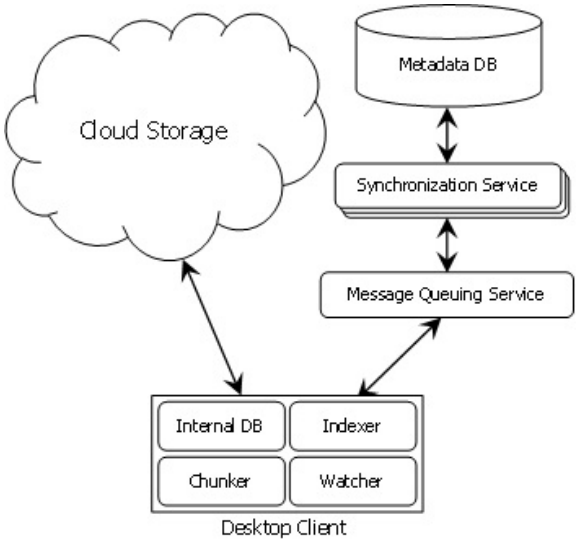

Figure 2. Logical Architecture

A. *Desktop Client*

The Desktop Client Application monitors the folders that are identified as workspace or sync folders and synchronizes them with the remote Cloud Storage. The Desktop Client interacts with the Synchronization Service to handle file metadata updates (e.g. file name, size, modification date, etc.). It also interacts with the backend Cloud Storage for storing the actual files.

Some of the most important requirements of the Desktop Client include upload and download of the files, detecting file changes in the sync folder, and handling conflicts due to offline or concurrent updates. The main components of the desktop client are Watcher, Chunker, Indexer, and Internal DB as described below.

- Watcher monitors the sync folders and notifies the Indexer of any action performed by the user for example when user create, delete, or update files or folders.
- Chunker splits the files into smaller pieces called chunks. To reconstruct a file, chunks will be joined back together in the correct order. A chunking algorithm can detect the parts of the files that have been modified by user and only transfer those parts to the Cloud Storage, saving on cloud storage space, bandwidth usage, and synchronization time.
- Indexer processes the events received from the Watcher and updates the internal database with information about the chunks of the modified files. Once the chunks are successfully submitted to the Cloud Storage, the Indexer will communicate with the Synchronization Service using the Message Queuing Service to update the Metadata Database with the changes.
- Internal Database keeps track of the chunks, files, their versions, and their location in the file system.

B. *Metadata Database*

The Metadata Database is responsible for maintaining the versioning and metadata information about files/chunks, users, and workspaces. The Metadata Database can be a relational database such as MySQL, or a NoSQL database service such as Amazon DynamoDB. Regardless of the type of the database, the Synchronization Service should be able to provide a consistent view of the files using a database, especially if more than one user work with the same file simultaneously. Since NoSQL data stores do not support ACID properties in favour of scalability and performance, we need to incorporate the support for ACID properties programmatically in the logic of our Synchronization Service in case we opt for this kind of databases. However, using a relational database can simplify the implementation of the Synchronization Service as ACID properties are natively supported by them. Selecting the type of the Metadata Database is a design decision that should be made early in the design phase of the system development life cycle.

The data that need to be stored in the Metadata Database include data about the chunks, objects, users, and their devices and workspaces (sync folders). The following key-value schema describes the persistence data structure of our cloud storage system in more details.

```
{
    “chunk_id”: “string”,
    “chunk_order”: “number”,
    “object”:
        {
        “version”: “number”,
        “is_folder”: “boolean”,
        “modified”: “number”,
        “file_name”: “string”,
        “file_extention”: “string”,
        “file_size”: “number”,
        “file_path”: “string”,
        “user”:
            {
            “user_name”: “string”,
            “email”: “string”,
            “quota_limit”: “number”,
            “quota_used”: number,
            “device”:
                {
                “device_name”: “string”
                “sync_folder”: “string”
                }
            }
        }
}
```

C. *Synchronization Service*

The Synchronization Service is the component that processes file updates made by a client and applies these changes to other subscribed clients. It also synchronizes clients' local databases with the information stored in the Metadata Database. The Synchronization Service is the most important part of the system architecture due to its critical role in managing the metadata and synchronizing users' files. Desktop clients communicate with the Synchronization Service to either obtain updates from the Cloud Storage, or send files and updates to the Cloud Storage and potentially other users. If a client was offline for a period of time, it polls the system for new updates as soon as it goes online. When the Synchronization Service receives an update request, it checks with the Metadata Database for consistency and then proceeds with the update. Subsequently, a notification is sent to all subscribed users or devices to report the file update.

As a design goal, the Synchronization Service should be designed in a way to transmit less data between clients and the Cloud Storage in order to achieve better response time. To meet this design goal, the Synchronization Service should employ a differencing algorithm to reduce the volume of the data that needs to be synchronized. Instead of transmitting entire files from clients to server or vice versa, most of the file synchronization algorithms just transmit the difference between two versions of a file. Therefore, only the part of the file that has been changed is transmitted. This also decreases bandwidth consumption and cloud data storage for the end user.

An essential part of the Synchronization Service is a synchronization algorithm. Rsync[5] is one of the most popular and high performance algorithms of this type that is able to compute the difference among different copies of data. Rsync partitions a file that is located on the server into several chunks with fixed block sizes, and uses a hash function to calculate its hash value to be sent to clients. The clients then use the hash values to determine whether to update the local copy of a chunk or not. Rsync is widely adopted by for data synchronization due to its simplicity and high performance.

To be able to provide an efficient and scalable synchronization protocol we consider using a communication middleware between clients and the Synchronization Service. The messaging middleware should provide scalable message queuing and change notification to support a high number of clients using pull or push strategies. This way, multiple Synchronization Service instances can receive requests from a global request queue, and the communication middleware will be able balance their load.

D. *Message Queuing Service*

An important part of our reference architecture is a messaging middleware that should be able to handle a substantial amount of reads and writes. A scalable Message Queuing Service that supports asynchronous message-based communication between clients and the Synchronization Service instances best fits the requirements of our application. The Message Queuing Service supports asynchronous and loosely coupled message-based communication between distributed components of the system. The Message Queuing Service should be of high performance, highly scalable, and be able to persistently store any number of messages in a highly available and reliable queue. The Message Queuing Service also provides load balancing and elasticity for multiple instances of the Synchronization Service.

Figre 3 illustrates two types of queues that are used in our Message Queuing Service. The Request Queue is a global queue that is shared among all clients. Clients' requests to update the Metadata Database through the Synchronization Service will be sent to the Request Queue. The Response Queues that correspond to individual subscribed clients are responsible for delivering the update messages to each client. Since a message will be deleted from the queue once received by a client, we need to create separate Response Queues for each client to be able to share an update message which should be sent to multiple subscribed clients.

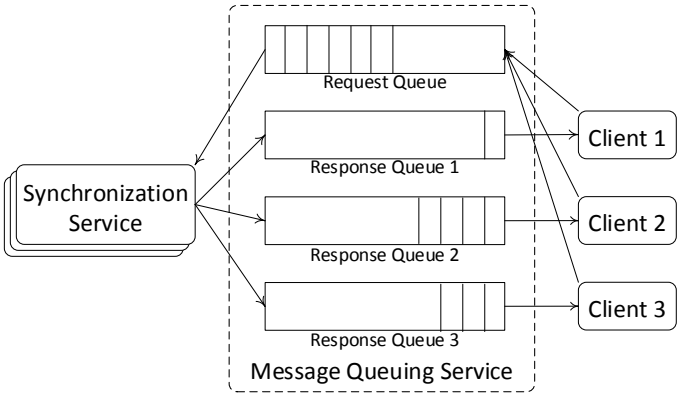

Figure 3. Message Queuing Service

E. Cloud Storage

Cloud Storage stores the chunks of the files uploaded by the users. Clients directly interact with the Cloud Storage to send and receive objects using the API provided by the cloud provider. The separation of the metadata from the object storage enables our reference architecture to use any Cloud Storage as the back-end data store.

F. File Processing

The sequence diagram in Figure 4 shows the interaction between the components of the application in a scenario when Client 1 updates a file that is shared with Client 2 and 3, so they should receive the update too. If the other clients were not online at the time of the update, the Message Queuing Service keeps the update notifications in separate response queues for them until they become online at a later time.

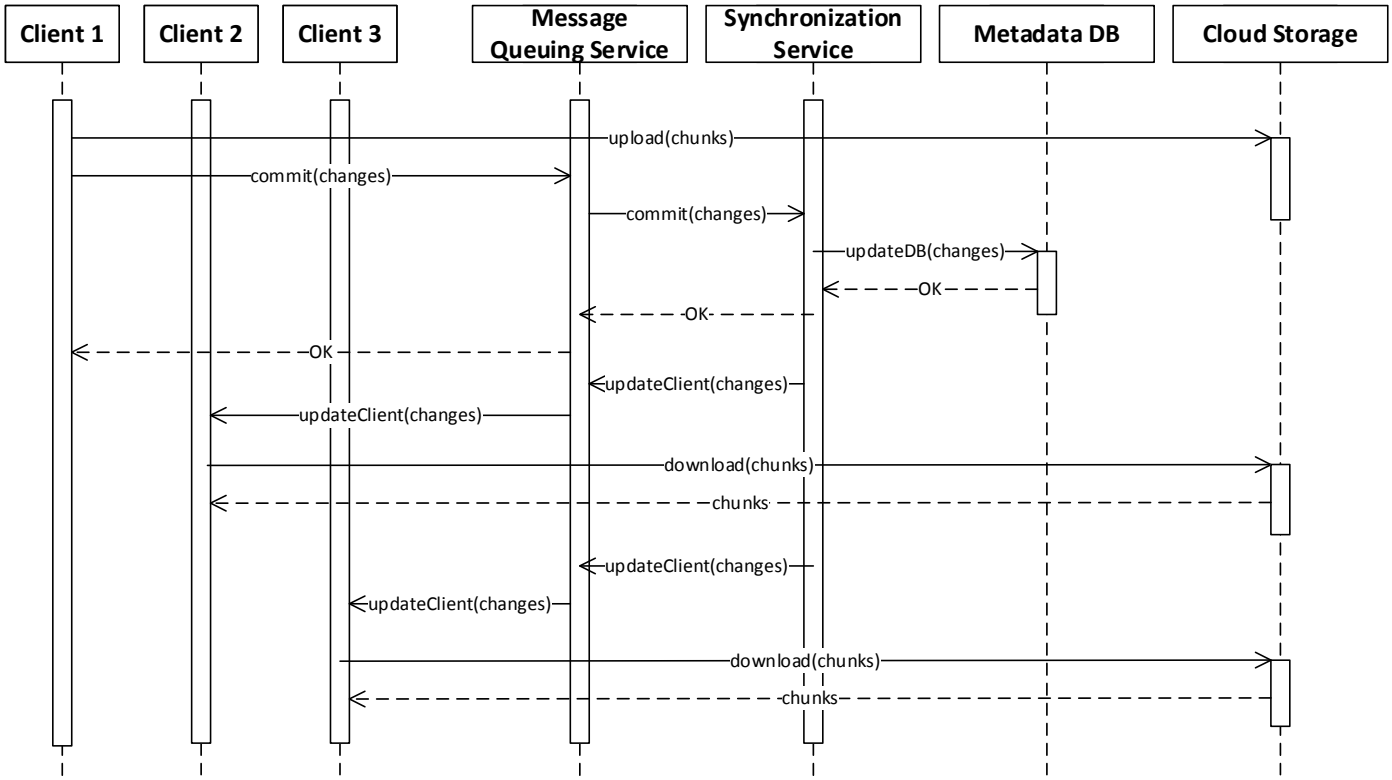

Figure 4. Sequence Diagram

IV. Cloud Deployment

For the purpose of this study, we chose Amazon as cloud service provider because it is currently one of the major players in the cloud computing market, with a proven maturity when compared to others, gathers all the services required by our system and offers data storage services at very low costs.

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and scalability.

Amazon offers compute, storage, databases and networking services in the Cloud, which are collectively known as Amazon Web Services (AWS). All services can be configured and monitored from a single web-based console, and are offered with a simple "pay-as-you-go" charging model. The most relevant services for our reference architecture are Amazon Simple Storage Service (S3), Amazon Simple Queue Service (SQS), Amazon Elastic Compute Cloud (EC2), and Amazon DynamoDB.

Amazon's S3, SQS and EC2 services are among the first utility computing services that are offered at a large scale. The "Success Stories" on Amazon's website describe some experiences using these services: NASA's Jet Propulsion Laboratory (JPL) uses Amazon EC2 to process high resolution satellite images that provide guidance and situational awareness to its robots. In addition, JPL relies on Amazon Cluster Compute Cloud Instances and Amazon SQS to deploy massive computations with less effort. 6 Waves Limited, a leading international publisher and developer of gaming applications on the Facebook platform, uses Amazon EC2 and Amazon S3 to host its social games with an audience of more than 50 million players per month. ElephantDrive turns to Amazon S3 to store client data, expanding their total amount of storage by nearly 20 percent each week while avoiding increased capital expenses [6].

Figure 5 depicts the overview of our deployment architecture which is in direct relationship with the logical

architecture presented in the previous section. Each component is explained in further detail below.

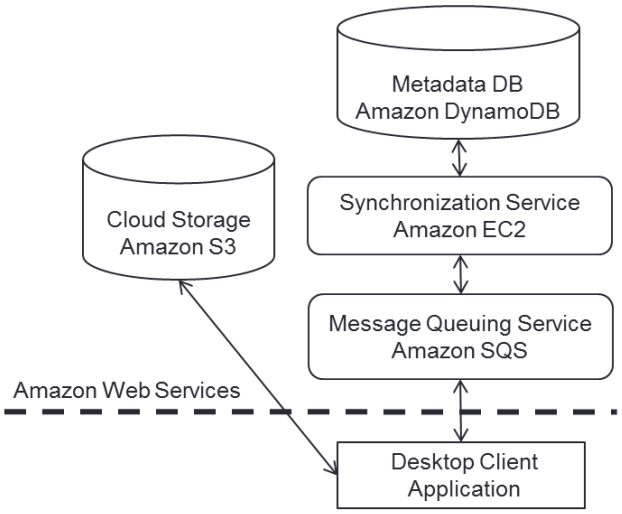

Figure 5. Deployment Architecture

A. Amazon S3 - Cloud Storage

Amazon S3 is a cloud storage service that offers software developers capabilities for storing large volumes of data, with additional features such as life cycle management and security. Amazon claims its service offers infinite storage capacity, unlimited data durability, 99.99% availability, and good data access performance. Amazon S3 is designed to make web-scale computing easier for developers. Amazon S3 provides a simple web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, fast, and inexpensive data storage infrastructure that Amazon uses to run its own global network of web sites [7].

Conceptually, Amazon S3 is an infinite store for objects of variable size (minimum 1 Byte, maximum 5 GB). An object is simply a byte container which is identified by a URI. Clients can read and update S3 objects remotely using a SOAP or REST-based interface; e.g., *get(uri)* returns an object and *put(uri, bytestream)* writes a new version of the object [8]. Each object is associated to a bucket. That is, when a user creates a new object, the user specifies into which bucket the new object should be placed.

Our Desktop Application directly interacts with Amazon S3 in order to store user files. Its integration with the operating system permits to monitor a local folder and synchronizes it with a remote bucket. When there is a change in the synchronized folder, the application receives a notification to process the event. Once the application identifies which file or files have been modified or created, it will proceed to store it in Amazon S3. In order to avoid any bottleneck, save traffic and storage costs, the application transfers only those parts of files (chunks) that have been modified or are new.

Amazon S3 provides different SDKs for easy integration with third party technologies. These kits simplify programming tasks by wrapping the underlying REST API. Our application would use the API called *com.amazonaws.services.s3.transfer* which provides a utility for managing transfers to Amazon S3. This method is a high level Java class called *TransferManager* that uploads data in parts using multiple connections and threads, and achieves the highest throughput in comparison with other popular S3 clients and programming libraries [9]. An example of how this method could be used in our system is shown in Appendix A.

B. Amazon SQS - Message Queuing Service

Amazon SQS is a highly reliable and scalable message delivery service that enables asynchronous message-based communication between the distributed components of larger-scale applications. Amazon SQS runs within Amazon's high-availability data centers, so queues will be available whenever are needed. To prevent messages from being lost or becoming unavailable, all messages are stored redundantly across multiple servers and data centers [10].

Amazon SQS can deliver unlimited number of messages at any time. The size of the message cannot be more than 256 KB. And it ensures at least 1 delivery of the message. It retains message up to 14 days. It also provides batching of messages up to 10 messages or 256 KB in total whichever is higher is applicable. When a message is received, it becomes locked while being processed. This keeps other application from processing the message simultaneously. If the message processing fails, the lock will expire and the message will be available again. In the case where the application needs more time for processing the lock timeout can be changed dynamically via the change message visibility operation [10].

In order to achieve scalability, our Desktop Application interacts with the Synchronization Service through Amazon SQS. Every time a change in the local folder is detected by the OS, our Desktop Application upload the affected chunks to Amazon S3 and communicates to the Synchronization Service in order to commit these changes to the metadata stored in DynamoDB. Furthermore, our Desktop Application receives notifications of committed changes from the Synchronization Service to apply them to the local folder.

As any other distributed system, Amazon SQS is used as message-passing mechanism between our components. This not only helps in making the different components loosely coupled, but also helps in building a more failure resilient system overall. If one component is receiving and processing requests faster than the other component (an unbalanced producer consumer situation), buffering will help make the overall system more resilient to bursts of traffic (or load). Amazon SQS also acts as a transient buffer between components. If a message is sent directly to a component, the receiver will need to consume it at a rate dictated by the sender. With message queues, sender and receiver are decoupled and the queue service smoothens out any "spiky" message traffic [11].

Thanks to our message-oriented communication, keeping the local folder in sync with the metadata storage is inexpensive, as any committed change is advertised as soon as

possible by means of asynchronous notifications. Amazon SQS in our architecture provides elasticity to distributed objects. It decouples sync control from the storage service, creates a transparent load balancing mechanisms, and it simplifies one-to-many communication.

However, a major limitation of Amazon SQS is that it does not guarantee delivering the messages exactly once. It guarantees delivery of the message at least once. That means there might be duplicate messages delivered to our system. The existence of duplicate messages comes from the fact that these messages are copied to multiple servers in order to provide high availability and increase the ability of parallel access [12]. According to Amazon, most of the time each message will be delivered exactly once. But in spite of that, it seems better to consider this aspect at the moment of building our components. In order to be able to verify duplications, our Desktop Application might use its internal database and our Synchronization Service might use DynamoDB. In both cases, these databases might be used to verify whether it is the first time that a message is processed or not. This technique might minimize any overhead caused by duplicate tasks execution.

Our Desktop Application and Synchronization Service would interact with Amazon SQS principally through the APIs: CreateQueue, DeleteQueue, SendMessage, ReceiveMessage and DeleteMessage. CreateQueue creates queues for use with AWS account. DeleteQueue deletes a queue. SendMessage adds messages to a specific queue. ReceiveMessage returns one or more messages from a queue. DeleteMessage removes a previously received message from a queue. Examples of calls to these methods can be found in Appendix B.

For the system security, we rely on the security of Amazon SQS. Authentication mechanisms are provided to ensure that messages stored in Amazon SQS queues are secured against unauthorized access. Only authorized users can access to the contents of queues. In order to keep the latency low, we would not add any encryption to messages.

C. Amazon EC2 - Synchronization Service

Amazon EC2 is a web service that provides resizable compute capacity in the Cloud. It is designed to make web-scale cloud computing easier for developers. Amazon EC2 offers a highly reliable environment where instances can be rapidly commissioned. The service runs within Amazon's proven network infrastructure and data centers. The EC2 Service Level Agreement commitment is 99.95% availability for each region [13].

Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. Instance types comprise varying combinations of CPU, memory, storage, and net-working capacity. In our architecture, the Synchronization Service runs on an instance type called "m3.medium" with 1 CPU Intel Xeon E5-2670 v2 (Ivy Bridge) and 3.75 GB RAM. The software pre-configured is the Amazon machine image called "JBoss powered by Bitnami" which comes with ready-to-run versions of Apache, MySQL and JBoss.

Our Synchronization Service is a key component in our middleware since it is in charge of managing the metadata to achieve data in sync. Desktop clients communicate with the Synchronization Service to obtain the changes occurred when they were offline, commit new versions of files, and receive events about remote modifications. Multiple instances might be running simultaneously in order to cope with this workload.

We take advantage of Amazon EC2 to meet the scalability requirements of our Synchronization Service. Amazon EC2 provides an Auto Scaling feature that permits define conditions to scale up and down an arbitrary group of instances. Scaling conditions can be defined to increase or decrease EC2 capacity by a certain percentage. In case of scaling up, when Auto Scaling detects that a condition has been met, it automatically adds the requisite amount of Amazon EC2 instances to the Auto Scaling Group. In case of scaling down, Auto Scaling decides which EC2 instance within the group is terminated once the scaling condition is met. When more than one instance meets this criterion, Auto Scaling will terminate the instance running for the longest portion of a billable instance-hour.

Auto Scaling can be integrated with Elastic Load Balancing to distribute incoming traffic across multiple EC2 instances. The Elastic Load Balancing service provides the required amount of load balancing capacity by routing traffic across instances within the Auto Scaling Group. It supports Amazon EC2 instances with any operating system and can perform load balancing by using diverse TCP ports and protocols.

EC2 instances for our Synchronization Service do not need to be exposed to the Internet since this component communicates only with Amazon SQS and Amazon DynamoDB. However, to extend security requirements, Amazon EC2 web interfaces allow specifying which groups may communicate with which other groups. This allows controlling access to instances in a highly dynamic environment.

D. Amazon DynamoDB - Metadata DB

Amazon DynamoDB is a fast and flexible NoSQL database service for all applications that need consistent, single-digit millisecond latency at any scale [14]. DynamoDB belongs to the category of key-value stores and is used to manage the state of several services of Amazon's e-commerce platform, which have high reliability requirements and need tight control over availability and performance.

DynamoDB uses a synthesis of well-known techniques to achieve scalability and reliability. Data is partitioned and replicated using hashing. Every node in the system is assigned to one or more points on a fixed circular space called "ring". Each data item, identified by a key, is assigned to a node by hashing its key with a hash function, whose output is a point on the ring, and then walking the ring clockwise to find the first node that appears on it (coordinator node). The consistency among replicas during updates is maintained by protocol similar to those used in quorum systems. This protocol has two parameters R/W - the minimal number of nodes that must participate in a successful read/write operation respectively. When a coordinator receives a write operation, it writes the data locally and then sends a write request to the other N-1

replica nodes. If at least W-1 nodes respond, the operation is considered successful and the coordinator informs the client. When a read operation is received, the coordinator sends a read request to the N-1 nodes, and if at least R-1 nodes respond, it returns the result to the client [15].

DynamoDB can provide the desired levels of availability and performance with a proper design and implementation, and can also handle successfully server and data center failures. DynamoDB is incrementally scalable and allows service owners to scale up and down based on their current request load [16].

In our reference architecture, DynamoDB is mainly responsible for storing all the metadata regarding files, versions, users and workspaces. The Synchronization Service interacts directly with DynamoDB to keep a consistent view of all this information. When a client connects to the system, the first thing it does is asking the Synchronization Service for changes that were made during the offline time period. When the Synchronization Service receives a commit operation of a file, it must first check that the metadata received is consistent. If the metadata is correct, it proceeds to save it to the database.

DynamoDB is a fully managed database service. One simply creates a database, sets throughput, and the service handle all the rest. Database operations can be done through AWS Management Console or the Amazon DynamoDB APIs. Our Synchronization Service would interact with DynamoDB by using the API methods called GetItem, PutItem and DeleteItem. The GetItem operation returns a set of attributes for an item that matches the primary key. The PutItem creates a new item, or replaces an old item with a new item (including all the attributes). The DeleteItem operator deletes a single item in a table by primary key. Some examples of how these methods could be used in our system are presented in Appendix C.

V. Performance and Cost Analysis

In this section, we are going to evaluate performance and cost of our proposed application. Performance is an integral part of our application and significant considerations must be put into technical design to achieve reasonable throughput. The application will be accessible at global level and should sustain large number of users.

Three scenarios are devised to analyze the performance and cost metrics of our cloud storage application. Variations in performance metrics and cost will be studied based on a set of initial assumptions and estimates that are required for our analysis. These assumptions and estimates are later defined in this section.

The application is supposed to be deployed on Amazon Web Services and we intend to theoretically calculate the cost of hosting the application on the Cloud based on our assumptions and estimates. Thereafter, operating cost will be compared against a proposed business model to analyze the cost-benefit of the application. In other words, we will determine the required pricing of the application to be profitable. The purpose of analyzing the cost and pricing is not to establish a business viable model but to develop a sense of cost of operating a cloud application with cloud computing and a performance that is achievable in reasonable cost limits.

A. Initial Assumptions

- Resources and cost analysis would be performed on one year baseline after application goes into production.
- Application provide services globally and available to all Internet users. This means that ideally application should have zero downtime 24x7.
- Application design baseline is to keep the UI simple and efficient. The data payload which will be exchanged on HTTP requests and response as well as size of HTTP requests and response, all inclusive will be 2MB each handshake (request, acknowledge request, response and payload). This can be split up by 500KB of payload and 100KB of request and response.
- The application business model will offer 2 broad categories of subscription. ‘Free’ storage up to specific storage capacity and ‘Paid’ storage for higher than ‘Free’ storage capacity.

B. Estimations

1) Estimate of subscribers after one year (signed-up users): Estimated first year user subscriptions are calculated as per initial ballpark number of subscriber and then growth in subscription based on reasonable (random) percentage to achieve a total hypothetical number of subscriptions at the end of first year. This method is more realistic scenario of user growth than just picking a random number as one year subscribers. After thirty users in first month, there is supposedly 15% monthly growth in number of users for three months, 20% for next four months and then 25% growth till the end of the year. Total user subscriptions, regardless of paid or free are 230. Compounding formula (Users = U * (1+r) ^t) can be used to calculate monthly growth where:

- U = monthly user subscription
- r = increase %
- t = months (1 for monthly)

Month	User	Growth (%)
1	30.0	0%
2	34.5	15%
3	39.7	15%
4	45.6	15%
5	54.8	20%
6	65.7	20%
7	78.8	20%
8	94.6	20%
9	118.3	25%
10	147.8	25%
11	184.8	25%
12	231.0	25%

Table 1

Total users at the end of year are 231. Hence, the resources will be acquired based on 231 signed-up (subscribed) users in first year. Performance parameters of acquired cloud resources and cost of resources will be calculated based on above estimate.

2) Estimate of upload & download (Data-in & Data-out) file size: The average file size of 10MB will be used to test the different scenarios. 10MB file uploaded/download will be used as baseline for estimating required resources for application performance testing. Cost will be calculated accordingly.

C. Cost of Acquired Resources

As per the architecture section, that computing resources acquired for the application are Amazon EC2 (computing) on-demand instance of m3.medium. Storage option for the application is Amazon S3 (persistent data storage). It is to be noted that Amazon EC2 instance also provide internal storage that can be used for storing meta-data and it does not exceed couple of gigabyte. Amazon EC2 instance's internal storage should be sufficient for it if utilized. This additional storage (meta-data) will not add in additional cost other than the cost of instance therefore is not considered in the scenarios and analysis.

Table 2 shows Amazon instance m3.medium and its bandwidth. These bandwidth measures are not advertised by Amazon and it largely depends on the external network infrastructure, distance of source server from the destination and the tier and class of server. However, based on independent tests, the conservative measure of average bandwidth output of the m3.medium instance is given in table 2 which is based on the experiment conducted by researchers as show in [17]. 391.00 mega-bits (Mb) is tested bandwidth which equals to 48.88 mega-byte (MB) per second (391.00/8). However, this certainly may not be the actual throughput received at the user's end which is degraded due to several network and external reasons. As a rule of thumb, we assume that 50% of actual tested bandwidth will be achievable, i.e. 24.45 MB/s. Table 2 also shows the monthly cost computed based on 732 hours a month.

Amazon EC2 instance (bandwidth advertised)	**Bandwidth**	**Unit Cost($)/hr**	**Cost/Month**
m3.medium	391.00 Mbps	$0.1330	$97.36
Max Bandwidth in MBps	48.88 MB/s		
Effective % of BW (achievable)	50%		
Effective BW (MBps)	24.4375 MB/s		

Table 2

For the Amazon S3 storage, the throughput in mega-byte (MB) per second is by tier and this is advertised by Amazon. For storage utilization greater than 1TB, the throughput is 50MB/s transfer out from the storage and we will consider 50% of it to be actual achievable of whichever tier the utilization will be. The actual tier wise throughput is shown in Table 3.

Amazon S3 Storage	**Throughput (MB/s)**	**Size (TB)**	**Cost/month/ GB**
1TB Tier	20.00	1.00	$0.0300
2-50 TB Tier	50.00	49.00	$0.0295
50-500 TB Tier	50.00	450.00	$0.0290
501-1000 TB Tier	50.00	500.00	$0.0285
Amazon S3 Data Transfer Out		**Size (TB)**	**Cost/month/ GB**
1TB (free)		1.00	$0.0900
Upto 10 TB		10.00	$0.0900
Upto 50 TB		40.00	$0.0900
Upto 150 TB		100.00	$0.0700

Table 3

D. Use Cases

1) Use Case 1: At the end of firs year, between 5%-50% of users from the total subscribers access the application in a day (0-24 hour period). The 24 hours will be divided into units of 2 hours. Therefore, there will be 12 units of measurement in a day. This is essentially a test based on optimal 'user traffic' in a day. It can also be extended to calculate performance and metrics for a month.

2) Use Case 2: Keeping the same condition as of Use Case 1, the file size of 10MB based on initial estimate will be increase to 50MB to test the performance and cost metrics. This is essentially stressing our initial 'data size' estimates and its impact on performance and cost.

3) Use Case 3: Keeping the user traffic conditions as Use Case 1 and file size of 10MB (also as per Use Case 1), user traffic will be increased in this case. This is essentially 'computing and network' test and its impact on overall performance and cost.

E. Performance Metrics of Resources

Little's Law will be extensively utilized to calculate the performance metrics. In brief, Little's law deals with the queueing theory and is an industry accepted method for calculating software performance metrics. Detailed derivation and explanation of Little's law is outside the scope of this paper. Little's law has following key variables:

1) Number of occurrences or Entities in the system [N]: Can be considered as number of simultaneous users accessing the application from perspective of our analysis.

2) Total Response Time or Average time entity spend in a system [R]: From perspective of our analysis, this is total of service time of HTTP request and response and data downloaded from storage. This also includes wait time. Therefore total response time consists of Service time of Request (A) + Service time of data download (B) + Wait time (W)

Total Response time = (A + B) + W

3) Throughput or Arrival Rate [T]: Throughput is number of user served. This will be calculated in the unit of per second.

The above key performance will be constrained by Utilization, Bandwidth (EC2 instance and S3 storage) and User-traffic.

F. Unit Calculation Using Available Data Variables

Number of total users are 231 (Table 1) and the request size is 500 KB + 100 KB + 100 KB = 700 KB per user. Data-request size (File size) = 10 MB as per the estimation set in the estimation section. Wait time can be assumed to be 0 seconds/ user to capture the impact of all users accessing the application resources at one time. Total response time to be calculated = (A + B) + W. Service time of request Size (A), service time of Data-request (B), and Wait time (W) will be added to the total of (A+B). The resources will be constrained by few factors such as CPU utilization to peak at 80%. Effective EC2 instance computing bandwidth is 24.45 MB/s as per Table 2. S3 storage resources are dynamically acquired according to the tier in effect as per Table 3.

First we will calculate the metrics for 1 user at first and then extend the trend with increasing use traffic based on percentage of total user over 24 hour period. The calculation shows that single user connecting on EC2 m3.medium instance (for http request) and downloading a 10MB file from S3 storage will have a total response time (including the think time of 2 sec) of 0.3868 seconds at 0.23% server utilization. Table 4 shows the basic estimates for single user.

Parameters/Inputs	**1 User (Unit Measure)**
Concurrent users (N)	1
Page size (KB)	500.00 KB
Http request (KB)	100 KB
Http response (KB)	100 KB
Total Request size(MB)	0.68 MB
Single file size to download (MB)	10 MB
Total file size in (MB) for all users (1 user)	10.00 MB

Table 4

G. Test for Use Cases (1,2 & 3)

1) Use Case 1:

Extending the unit calculation, if 5% to 60% of subscribers will access the application simultaneously in a timeframe of 1-2 hours a day, the throughput achieved will be between 0.8 – 1.6391 service requests per second. Refer to Table-A in Appendix for the actual data points and working, it also shows the bottle-neck and the need for additional instances (required server resources) when single server is bottle-necked. The data in the appendix table has been expanded to show the increased scalability and need for additional computing resources. The actual graphs are shown below. Average response time varies between minimum of 2 seconds and the maximum of 8 seconds, however think time (Z) is the most impactful component to the overall response time. Think time keeps the overall performance analysis and calculations realistic and reasonable. If think time is set to zero (0) this would mean that all users send the request to the server at the same time without any queue, pushing the performance metrics to non-realistic outputs. Service time of computing component (i.e. EC2 instance) does not have much impact as the request sizes and meta-data size are not too big compared to data size, thus have minimum impact on average response time. Data transfer service time primarily impacts the average response time.

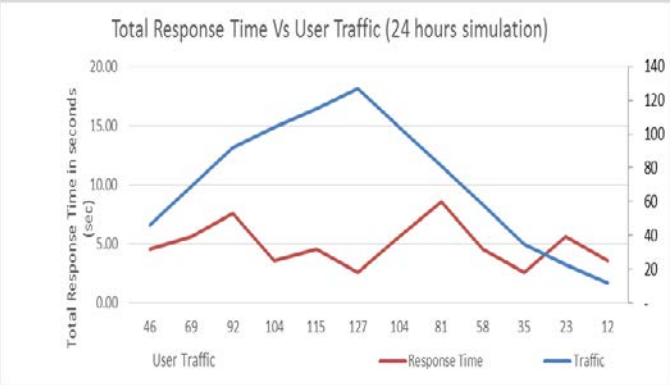

Figure 6. Total Response Time vs User Traffic

In Table-A in Appendix the server utilization is shown in the last row where highlighted instances are utilized above 95% or close to 100%. This is a saturation point and server bottleneck conditions. To overcome this, EC2 instance needs to be configured to scale out and add an extra EC2 instance to the application. Once the additional instance is up, throughput will be reduced for that time frame and utilization will be distributed over two EC2 instances. Table-B in the Appendix shows throughput and utilization split across two EC2 instances. Highlighted instances shows the hours when the second instances of EC2 is added to the application due to high traffic volume and the utilization threshold is set to be 80%. As soon as instance utilization reaches 80%, the additional instance is enabled and up. The graphs show the impact of added instance and sudden drop and spike in throughput and utilizations.

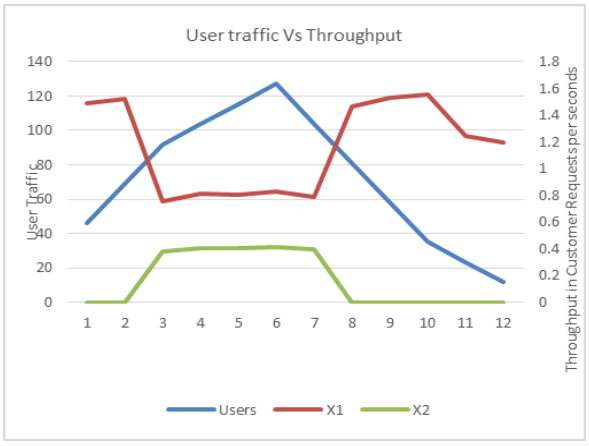

Figure 7. Use Traffic vs Throughput

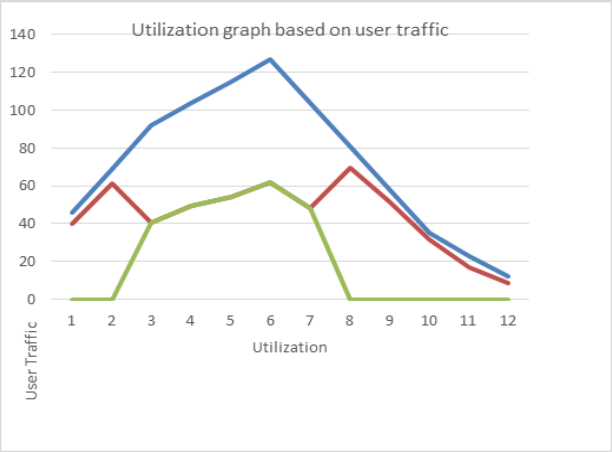

Figure 8. Utilization Graph Based on User Traffic

Cost of resouces:

Based on assumption that this daily pattern continues and is considered normal application usage, the monthly cost of resources will be as follows

Final total cost of compute resources (Amazon EC2) for 1 month is $137.92. Refer to Table-C in Appendix.

From the Table-A in Appendix total data downloaded in a day can be calculated for a month. This information is summarized in Table 5 below.

	24 hours	Month
Total data downloaded (MB)	8660.000	
(GB)	8.457	**257.939 GB**
(TB)	0.008	**0.252 TB**

Table 5

Assuming that overall data storage is still under 1TB, the cost of storage (only) charged separate by Amazon S3 will be $7.74. Cost of data transfer out is per month per GB, therefore it will be $23.31 for a month while data-in (upload) is free. Refer to Table-D (i) and (ii) in Appendix. Cumulative total cost of data storage and data transfer out is $30.95. The total cost of compute, data storage and data transfer out will be $137.92 + $30.95 equals to $168.87 per month.

Resource	Monthly Cost Use Case 1
Compute cost	$137.92
Storage & Data transfer cost	$30.95
Grand Total	**$168.87**

Table 6

We will perform tests on other two use cases by stressing the variables as described above in the paper and show graphical outputs and impact on cost.

2) Use Case 2:

Use case 2 is actually stressed by file size. The file size has been increased five folds to 50 MB. There is no significant impact on the average response time in terms of absolute time, a mere 2-3 seconds additional time however relatively there is an increase of approximately. 80% on minimum and around 30% increase in maximum average response time as shown in Table 7 below.

Average response time	Use case 1	Use case 2
Min	2.59 sec	4.87 sec
Max	8.59 sec	10.87 sec

Table 7

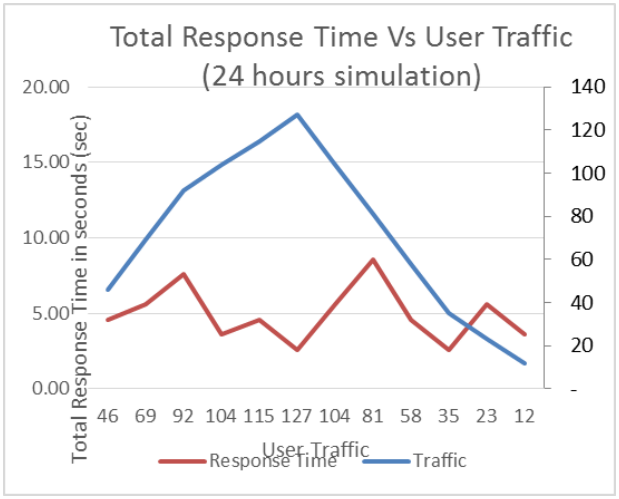

Figure 9. Total Response Time vs User Traffic

Throughput has a drastic impact due to increase in the data file size but the average response time is sustained due to additional instance deployed and both running at high utilization.

Throughput	Use Case 1	Use Case 2
Min	1.134 req/sec	0.254 req/sec
Max	1.556 req/sec	0.341 req/sec
Utilization	**Use Case 1**	**Use Case 2**
Min	8.41%	11.04%
Max	123.67%	126.31%

Table 8

In Table 8 above, utilization more than 100% is the sum of total utilization on both instances. To know the actual utilization of each instance, greater than 100% utilization should be divided by 2.

Cost of resources:

The most impact of increasing the file size is on cost rather than performance. Again, the computing impact is minimal as request size and meta-data size is not the primary impactor hence the cost of computing or Amazon EC2 instance is unchanged as in Use case 1 the instances were running on low utilization and use case 2 only the utilization has increased on

the instances but they still scale for same number of hours. However, the major impact is on data transfer-out cost which is increased from $30.95 to $292.55 per month.

Resource	Monthly Cost Use Case 2
Compute cost	$137.92
Storage & Data transfer cost	$154.63
Grand Total	**$292.55**

Table 9

3) Use Case 3:

In this use case scenario we are going to assume that user traffic increases over time and performance and cost metrics analysis will be performed. From initial one year baseline total of 231 subscribers, we will double it to 460 users/subscribers. Increased number of user has direct impact on computing resources as throughput and utilization increases pushing utilization at each of two EC2 instance to beyond 100%. This means that third instance would be required. Table-E in Appendix is the utilization table which shows the three instances enabled during the hours of day and how many hours each instance is enabled.

Throughput	Use Case 1	Use Case 2	User Case 3
Min	1.134 req/sec	0.254 req/sec	1.190 req/sec
Max	1.556 req/sec	0.341 req/sec	1.640 req/sec
Utilization	**Use Case 1**	**Use case 2**	**Use Case 3**
Min	8%	11 %	19 %
Max	124 %	126 %	250%

Table 10

Cost of resources:

Based on this scalability the cost of Amazon EC2 instance will increase to $340.75 shown in Table 11 below.

EC2 instance	Enable	Cost($)/hr	Hours Up	Cost($)/Day
m3.medium	*Y*	$0.1330	24.00	$ 3.19
m3.medium	*Y*	$0.1330	20.00	$ 2.66
m3.medium	*Y*	$0.5320	10.00	$ 5.32
				$ 340.75

Table 11

Data transfer out cost will proportionately be double that of use case 1 as a result of subscribed user base doubled.

Resource	Monthly Cost Use Case 3
Compute cost	$340.75
Storage & Data transfer cost	$61.66
Grand Total	**$402.40**

Table 12

H. Evaluation of running cost and proposed revenue model

From the three use cases discussed above, below is the summary of cost for each use case. It is apparent that cost increase in use case 2 is attributed to increased data transfer out while increased cost in use case 3 is attributed to bigger user base. It is to note that five times increase of data transfer out from 10MB in use case 1 (optimal scenario) to 50MB in use case 2 hiked the cost up to 75% while twice the increase in user base (231 to 460) increased the cost by 137%

Resource	Monthly Cost Use Case 1	Monthly Cost Use Case 2	Monthly Cost Use Case 3
Compute	$137.92	$137.92	$340.75
Storage & Data transfer	$30.95	$154.63	$61.66
Grand Total	**$168.87**	**$292.55**	**$402.40**

Table 13

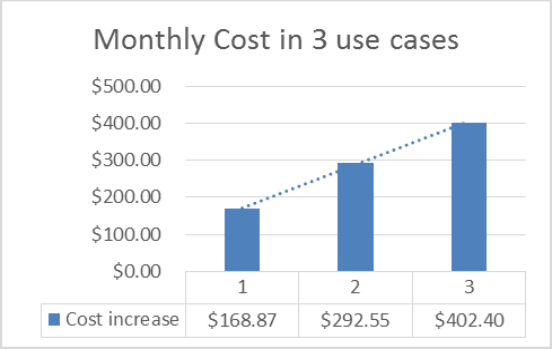

Figure 10. Monthly Cost in All Three Use cases

A simple proposed business model for storage capacity to the end-user is shown below.

Price for storage capacity

- Each subscribed user (signed-up user) will be given first 15GB free of cost.
- 15GB + additional storage will come at a cost
- 15GB – 100 GB storage capacity will have a flat cost (to be determined).
- 100GB – 500GB storage capacity will have a flat cost (to be determined).
- 500GB – 1TB storage capacity will have a flat cost (to be determined).
- 1TB is maximum storage capacity limit per user.

Based on the above proposed price offering model, it would require less than 10% subscriber to be service paying users. E.g. 17 users will be required on a $10 per 15GB plan to break-even the operating cost of application. 17 users are actually 8% of total subscribed users.

	Number of paid users required			% of total base		
Paid users Required to break even	**Use Case 1**	**Use Case 2**	**Use Case 3**	**Use Case 1**	**Use Case 2**	**Use Case 3**
$10/15GB slot	17	29	40	8%	13%	9%
$15/15GB slot	11	20	27	5%	9%	6%
$20/15GB slot	8	15	20	4%	7%	4%

Table 14

VI. Conclusion

In this paper, we have presented reference architecture and design goals of a Dropbox-like file storage system by identifying its subsystems, components, communication protocols, and persistence mechanism. Our architecture relies on a loosely-coupled asynchronous communication framework for providing elasticity and load balancing to distributed objects using message queuing.

Regarding system deployment, we have identified the physical cloud infrastructure and technologies that can be used for the implementation of our solution. Amazon's platform is built for high availability, reliability and efficiency. Services like Amazon S3 and EC2 seem to be the right choice for deployment of our system in the Cloud. However, to take advantage of Amazon SQS and DynamoDB services we need further study.

We have evaluated the performance and cost benefits of creating this service on the cloud. The analysis shows that the application could be profitable if the assumptions and estimates are sustainable in real conditions. This analysis shows that the application could be reasonably profitable if the assumptions and estimates are sustainable in actual (real world) conditions. The pricing here is also competitive to other cloud storage services being offered on the Internet. Even if other support and over-heads costs are considered, the margin of profitability can be sustained as only ~10% users are required to cover infrastructure cost and few more percentage points can cover other costs as well. Cost of storage only is cheapest portion of the cost as long as user base uses storage nominally. To give an idea about the storage cost, if all 231 users in optimal use case utilize 15B of free storage that comes to about 3.5TB (231 x 15GB = 3465 GB / 1024 = 3.3 TB) will cost approximately $130 on Amazon S3. This cost is not hard to cover if the subscription base has reasonable paid users of the service.

References

[1] I. Drago, M. Mellia, M. Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside dropbox: Understanding personal cloud storage services. In Proc. of ACM IMC, pages 481–494, 2012.

[2] Librsync. (n.d.). Retrieved April 18, 2015, from https://github.com/librsync/librsync

[3] Li, Z., Wilson, C., Jiang, Z., Liu, Y., Zhao, B. Y., Jin, C., ... & Dai, Y. (2013). Efficient batched synchronization in dropbox-like cloud storage services. In Middleware 2013 (pp. 307-327). Springer Berlin Heidelberg.

[4] Lopez, P. G., Sanchez-Artigas, M., Toda, S., Cotes, C., & Lenton, J. (2014, December). Stacksync: Bringing elasticity to dropbox-like file synchronization. In Proceedings of the 15th International Middleware Conference (pp. 49-60). ACM.

[5] Tridgell, A., & Mackerras, P. (1996). The rsync algorithm.

[6] All AWS Case Studies. (n.d.). Retrieved May 3, 2015, from http://aws.amazon.com/solutions/case-studies/all/

[7] AWS | Amazon Simple Storage Service (S3) - Online Cloud Storage for Data & Files. (n.d.). Retrieved May 3, 2015, from http://aws.amazon.com/s3/

[8] Brantner, M., et al, "Building a Database on S3", Proceedings of the 2008 ACM SIGMOD international conference on Management of data.

[9] Hobin Yoon, et al, "Interactive Use of Cloud Services: Amazon SQS and S3", IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2012.

[10] AWS | Amazon Simple Queue Service - Hosted Message Queuing Service. (n.d.). Retrieved May 3, 2015, from http://aws.amazon.com/sqs/

[11] Popovic, K., et al, "REST-style Actionscript programming interface for message distribution using Amazon Simple Queue Service", MIPRO, 2012 Proceedings of the 35th International Convention.

[12] Sadooghi, I., et al, "Achieving Efficient Distributed Scheduling with Message Queues in the Cloud for Many-Task Computing and High-Performance Computing", IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2014.

[13] AWS | Amazon Elastic Compute Cloud (EC2) - Scalable Cloud Hosting. (n.d.). Retrieved May 3, 2015, from http://aws.amazon.com/ec2/

[14] AWS | Amazon DynamoDB - NoSQL Cloud Database Service. (n.d.). Retrieved May 3, 2015, from http://aws.amazon.com/dynamodb/

[15] Weintraub, G., "Dynamo and BigTable - Review and Comparison", 2014 IEEE 28-th Convention of Electrical and Electronics Engineers in Israel.

[16] DeCandia, G., et al, "Dynamo: Amazon's Highly Available Key-value Store", Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, 2007.

[17] Benchmarking: Network Performance of m1 and m3 instances using iperf tool. (n.d.). Retrieved May 2, 2015, from http://blog.flux7.com/blogs/benchmarks/benchmarking-network-performance-of-m1-and-m3-instances-using-iperf-tool

[18] AWS | Amazon EC2 | Pricing. (n.d.). Retrieved May 3, 2015, from http://aws.amazon.com/ec2/pricing/

Appendices

A. Amazon S3 - Use of Programming Libraries
B. Amazon SQS - Use of Programming Libraries
C. DynamoDB - Use of Programming Libraries
D. User traffic pattern over a 24 hours period
E. Throughput & Utilization while extra instances are added to accommodate high traffic
F. Cost of computing Amazon EC2 instance
G. Cost of Amazon S3 data storage
H. Cost of Amazon S3 data transfer out
I. Utilization of the three instances enabled during the hours of day

APPENDICES

A. *Amazon S3 - Use of Programming Libraries*

The package *com.amazonaws.services.s3.transfer* included in the AWS SDK for Java provides the *TransferManager* class which is high level utility for managing transfers to Amazon S3. When possible, *TransferManager* attempts to use multiple threads to upload multiple parts of a single upload at once. When dealing with large content sizes and high bandwidth, this can have a significant increase on throughput.

Uploading data

```
AWSCredentials myCredentials = new BasicAWSCredentials(...);
TransferManager tx = new TransferManager(myCredentials);
Upload myUpload = tx.upload(myBucket, myFile.getName(), myFile);

// We can poll our transfer's status to check its progress
if (myUpload.isDone() == false) {
     System.out.println("Transfer: " + myUpload.getDescription());
     System.out.println("  - State: " + myUpload.getState());
     System.out.println("  - Progress: " + myUpload.getProgress().getBytesTransferred());
}

// Transfers also allow us to set a <code>ProgressListener</code> to receive
// asynchronous notifications about your transfer's progress.
myUpload.addProgressListener(myProgressListener);

// Or we can block the current thread and wait for our transfer
// to complete. If the transfer fails, this method will throw an
// AmazonClientException or AmazonServiceException detailing the reason.
myUpload.waitForCompletion();

// After the upload is complete, we call shutdownNow to release the resources.
tx.shutdownNow();
```

B. *Amazon SQS - Use of Programming Libraries*

The following examples illustrate how our system could realize several operations in Java with a queue.

Creating a queue

```
System.out.println("Creating a new SQS queue called Sync_Queue.\n");
CreateQueueRequest createQueueRequest = new reateQueueRequest().withQueueName("Sync_Queue");
String myQueueUrl = sqs.createQueue(createQueueRequest).getQueueUrl();
```

Sending a message

```
System.out.println("Sending a message to Sync_Queue.\n");
sqs.sendMessage(new SendMessageRequest().withQueueUrl(myQueueUrl).withMessageBody("This is my
   message text."));
```

Receiving a Message

```
System.out.println("Receiving messages from Sync_Queue.\n");
ReceiveMessageRequest receiveMessageRequest = new ReceiveMessageRequest(myQueueUrl);
List<Message> messages = sqs.receiveMessage(receiveMessageRequest).getMessages();
for (Message message : messages) {
  System.out.println(" Message");
  System.out.println(" MessageId: " + message.getMessageId());
  System.out.println(" ReceiptHandle: " + message.getReceiptHandle());
  System.out.println(" MD5OfBody: " + message.getMD5OfBody());
  System.out.println(" Body: " + message.getBody());
    for (Entry<String, String> entry : message.getAttributes().entrySet()) {
      System.out.println(" Attribute");
```

```
12:       System.out.println(" Name: " + entry.getKey());
13:       System.out.println(" Value: " + entry.getValue());
14:     }
15: }
16: System.out.println();
```

C. DynamoDB - Use of Programming Libraries

The following Java code snippet exemplifies how our solution could perform database operations over DynamoDB.

Getting an item

```
1: DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(new ProfileCredentialsProvider()));
2: Table table = dynamoDB.getTable("Metadata");
3: Item item = table.getItem("Chunk_Id", 101);
```

Putting an item

```
 1: DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(new ProfileCredentialsProvider()));
 2: Table table = dynamoDB.getTable("Metadata");
 3: // Build the item
 4: Item item = new Item()
 5:    .withPrimaryKey("Chunk_Id", 206)
 6:    .withString("Chunk_Order", "21")
 7:    .withString("Is_Folder", "0")
 8:    .withString("File_Name", "Project_A")
 9:    .withString("File_Extension", "doc")
10: // Write the item to the table
11: PutItemOutcome outcome = table.putItem(item);
```

Deleting an item

```
1: DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(new ProfileCredentialsProvider()));
3: Table table = dynamoDB.getTable("Metadata");
4: DeleteItemOutcome outcome = table.deleteItem("Chunk_Id", 101);
```

D. Based on user traffic pattern over a 24 hours period, the data is shown in the below table.

Parameters/Inputs	**20%**	**30%**	**40%**	**45%**	**50%**	**55%**	**45%**	**35%**	**25%**	**15%**	**10%**	**5%**
Number of concurrent users (N)	46	69	92	104	115	127	104	81	58	35	23	12
Page size (KB)	500 KB											
Http request (KB)	10 KB											
Http response (KB)	10 KB											
Total Request/Response size(MB)	31.45	47.17	62.89	71.09	78.61	86.82	71.09	55.37	39.65	23.93	15.72	8.2
Single file size to download (MB)	10 MB											
Total file data (MB) for all number of users	460 MB	690 MB	920 MB	1040 MB	1150 MB	1270 MB	1040 MB	810 MB	580 MB	350 MB	230 MB	120 MB
Hours of day	**1-2**	**3-4**	**5-6**	**7-8**	**9-10**	**11-12**	**13-14**	**15-16**	**17-18**	**19-20**	**21-22**	**23-24**
Service Time (A) (sec)	0.643	0.965	1.287	1.455	1.608	1.776	1.455	1.133	0.811	0.490	0.322	0.168
Service Time (B) (sec)	26.286	39.429	52.571	59.429	65.714	72.571	59.429	46.286	33.143	20.000	13.143	6.857
Wait Time (W) (sec)	0	0	0	0	0	0	0	0	0	0	0	0.000
Total Service Time (D) = (A+B) + W	26.929	40.394	53.858	60.883	67.323	74.348	60.883	47.419	33.954	20.490	13.465	7.025
Average think time (Z) seconds	4.00	5.00	7.00	3.00	4.00	2.00	5.00	8.00	4.00	2.00	5.00	3.000
Total Response Time [R] = D + Z (sec)	**210.929**	**385.394**	**697.858**	**372.883**	**527.323**	**328.348**	**580.883**	**695.419**	**265.954**	**90.490**	**128.465**	**43.025**
Average Response Time [R avg] = [Rt]/N	4.585	5.585	7.585	3.585	4.585	2.585	5.585	8.585	4.585	2.585	5.585	3.585
Total Throughput (X) [Requests(N) / sec]	**1.487**	**1.520**	**1.512**	**1.628**	**1.612**	**1.663**	**1.579**	**1.462**	**1.528**	**1.556**	**1.246**	**1.197**
Utilization U= X * D [%]	40.05	61.40	81.42	**99.12**	**108.55**	**123.67**	**96.11**	69.31	51.89	31.89	16.77	8.41

Table -A

E. Throughput & Utilization while extra instances are added to accommodate high traffic

Hours of day	**1-2**	**3-4**	**5-6**	**7-8**	**9-10**	**11-12**	**13-14**	**15-16**	**17-18**	**19-20**	**21-22**	**23-24**
Total Throughput [Rt/sec]	**1.487**	**1.520**	**1.512**	**1.628**	**1.612**	**1.663**	**1.579**	**1.462**	**1.528**	**1.556**	**1.246**	**1.197**
Throughput X1 (Req/sec)	1.487	1.520	0.756	0.814	0.806	0.832	0.789	1.462	1.528	1.556	1.246	1.197
Throughput X2 (Req/sec)	0.000	0.000	0.378	0.407	0.403	0.416	0.395	0.000	0.000	0.000	0.000	0.000
Utilization U= X * D [%]	40.05	61.40	81.42	**99.12**	**108.55**	**123.67**	**96.11**	69.31	51.89	31.89	16.77	8.41
Utilization U1 (%)	40.0%	61.4%	40.7%	49.5%	54.2%	61.8%	48.0%	69.3%	51.8%	31.8%	16.7%	8.4%
Utilization U2 (%)	0.000	0.000	40.7%	49.5%	54.2%	61.8%	48.0%	0.000	0.000	0.000	0.000	0.000

Table -B

F. Cost of computing Amazon EC2 instance

EC2 instance	**BW**	**Enable**	**Cost($)/hr**	**Hours Up**	**Cost($)/Day**
m3.medium	391.00	*Y*	$0.1330	24.00	$ 3.19
m3.medium	391.00	*Y*	$0.1330	10.00	$ 1.33
Max Bandwidth in (declared)	97.75 MB/s				
Effective % of BW (achievable)	50%				
Effective BW (MBps)	48.875 MB/s				
Total Monthly Cost					**$ 137.92**

Table -C

G. *Cost* of Amazon S3 data storage

Amazon S3 Storage	Throughput (MBps)	Enable	Size (TB)	Cost($)/month/GB	Data Distribution (TB)	Cost($)/Month
1TB Tier	20	1	1	0.03	0.252	$ 7.74
2-50 TB Tier	50	0	49	0.0295	0.000	$ -
50-500 TB Tier	50	0	450	0.029	0.000	$ -
501-1000 TB Tier	50	0	500	0.0285		$ -
Throughput latency	50%					
Actual Average Throughput	17.5					
Total Monthly Cost						**$ 7.74**

Table –D (i)

H. Cost of Amazon S3 data transfer out

Amazon S3 Data Transfer out		Enable	Size (TB)	Cost($)/month/GB	Data Distribution (TB)	Cost($)/Month
1TB		1	1	0.09	0.251893997	$ 23.21
Up to 10 TB		0	10	0.09	0	$ -
Up to 50 TB		0	40	0.09	0	$ -
Up to 150 TB		0	100	0.07		$ -
Total Monthly Cost						**$ 23.21**
Cumulative cost of data storage and data out						**$ 30.95**

Table –D (ii)

I. Utilization table shows the three instances enabled during the hours of day and how many hours each instance is up.

Hours of day	1-2	3-4	5-6	7-8	9-10	11-12	13-14	15-16	17-18	19-20	21-22	23-24	Hours/Day
Total utilization	86	130	173	202	223	250	199	148	109	66	39	19	
Instance 1	x	x	x	x	x	x	x	x	X	66	39	19	**24**
Instance 2	x	x	x	x	x	x	x	x	X				**18**
Instance 3			x	x	x	x	x						**10**

Table –E