Int. J. Communications, Network and System Sciences, 2017, 10, 59-74
http://www.scirp.org/journal/ijcns
ISSN Online: 1913-3723
ISSN Print: 1913-3715

New Synchronization Algorithm Based on Delta Synchronization for Compressed Files in the Mobile Cloud Environment

Rizik M. H. Al-Sayyed, Feras F. Namous, AlMonther H. Alkhalafat, Bashar Al-Shboul, Samar Al-Saqqa[*]

Department of Business Information Technology, King Abdulla II School for Information Technology, The University of Jordan, Amman, Jordan
Email: r.alsayyed@ju.edu.jo, fra8150150@fgs.ju.edu.jo, alm8150570@fgs.ju.edu.jo, b.shboul@ju.edu.jo, *s.alsaqqa@ju.edu.jo
s.alsaqqa@ju.edu.jo

How to cite this paper: Al-Sayyed, R.M.H., Namous, F.F., Alkhalafat, A.H., Al-Shboul, B. and Al-Saqqa, S. (2017) New Synchronization Algorithm Based on Delta Synchronization for Compressed Files in the Mobile Cloud Environment. *Int. J. Communications, Network and System Sciences*, **10**, 59-74.
https://doi.org/10.4236/ijcns.2017.104004

Received: November 19, 2016
Accepted: April 25, 2017
Published: April 28, 2017

Abstract

The fast growing market of mobile device adoption and cloud computing has led to exploitation of mobile devices utilizing cloud services. One major challenge facing the usage of mobile devices in the cloud environment is mobile synchronization to the cloud, e.g., synchronizing contacts, text messages, images, and videos. Owing to the expected high volume of traffic and high time complexity required for synchronization, an appropriate synchronization algorithm needs to be developed. Delta synchronization is one method of synchronizing compressed files that requires uploading the whole file, even when no changes were made or if it was only partially changed. In the present study, we proposed an algorithm, based on Delta synchronization, to solve the problem of synchronizing compressed files under various forms of modification (e.g., not modified, partially modified, or completely modified). To measure the efficiency of our proposed algorithm, we compared it to the Dropbox application algorithm. The results demonstrated that our algorithm outperformed the regular Dropbox synchronization mechanism by reducing the synchronization time, cost, and traffic load between clients and the cloud service provider.

Keywords

Mobile Cloud Computing, Delta Synchronization, Compressed Files, Dropbox

1. Introduction

Mobile devices play an essential role in everyday life owing to their communication effectiveness and mobility features. Mobile companies compete to build business models based on developing mobile applications that can assist people and dominate the marketplace.

Various mobile platforms (e.g., ios and Android) compete to enrich their users with powerful applications that might include cloud applications.

Cloud computing has been widely recognized as the next generation in computing infrastructure; it offers many advantages such as allowing users to employ infrastructure devices (*i.e*., servers, networks, and storages), platforms (*i.e*., middleware services and operating systems), and software (*i.e*., application programs) provided by cloud providers (e.g., Google, Amazon, Azure, and Oracle) at low cost. In addition, cloud computing enables users to utilize resources in an on-demand manner. As a result, both academia and the IT industry have been encouraged to improve and merge mobile computing and cloud computing together into mobile cloud computing (MCC) and provide possible solutions for any challenges associated with this new environment. One of these challenges is mobile cloud synchronization [1], which is the focus of the present study. For mobile cloud synchronization, many IT companies are competing to determine the best solutions for any associated problems and are trying to determine the type of data that the user requires to be portable.

MCC is a combination of mobile computing, cloud computing, and mobile internet, and it integrates the advantages of these three technologies. Therefore, it can be called cloud computing for mobiles.

At its simplest, MCC refers to an infrastructure where data storage and data processing are both performed from a mobile device, where applications transfer the computing power and data storage onto the cloud [2]. Therefore, various business opportunities for mobile network operators, as well as cloud providers, arise as the large computing power of the cloud is harnessed to benefit different types of mobile users.

As shown in **Figure 1**, the architecture of mobile cloud computing consists of two main components: mobile computing and cloud computing [3]. The mobile computing component consists of various mobile devices such as smart phones, PDAs, and laptops. These devices are normally connected to a network via different communication technologies that enable the mobile user to send requests to the cloud service provider so adequate resources can be allocated to the mobile devices throughout the established connection. By starting the web application, the monitoring and calculating functions of the system are implemented to guarantee that quality of service is maintained until the connection is completed and the task is performed. This process includes accomplishing tasks such as sending responses rapidly, synchronizing files, and load balancing to ensure that the resources are allocated fairly to the appropriate clients.

MCC has a major benefit of moving data storage and computing power from mobile devices to the cloud, with other benefits including solving the problem of short battery life via transferring the execution of extensive communications to the cloud and centralizing security that might be useful in detecting patterns of security threats.

However, many challenges arise for MCC such as synchronization, quality of communications (*i.e.*, wired, wireless, 3G, and LTE), quality of service (*i.e.*, latency, delay, and bandwidth), mobile device classification (*i.e.*, high-end, low-end, battery, CPU, and RAM), and individualization of mobile for various available operating systems.

In this context, we need to introduce cloud storage. Figure 2 briefly illustrates the evolution of cloud storage that is based on the traditional network storage and on hosted storage [4]. Applications employ traditional network storage to

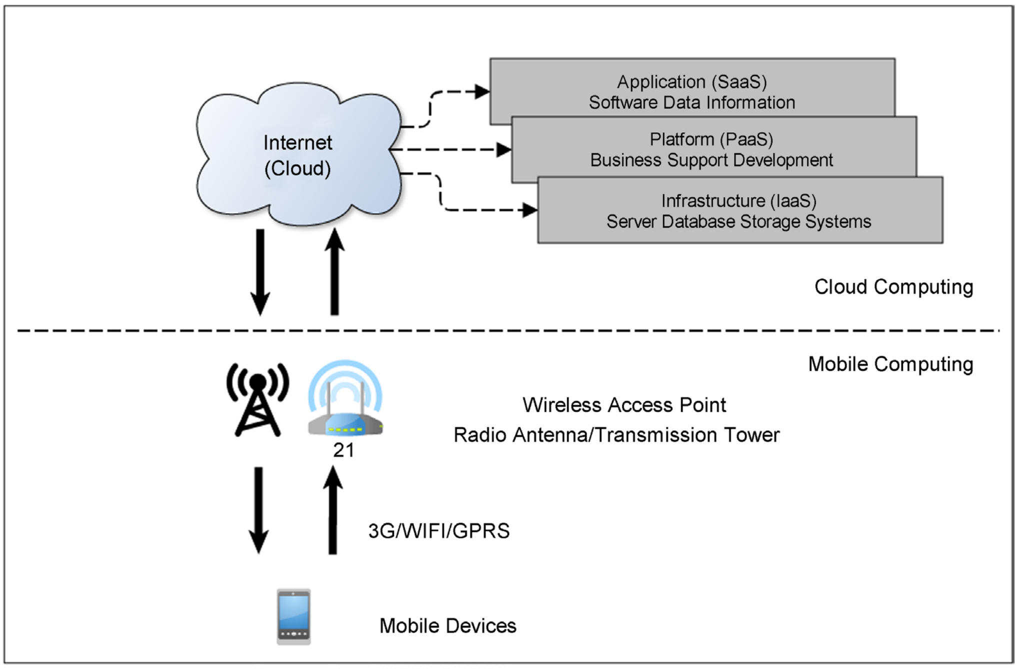
Internet (Cloud)
Application (SaaS) Software Data Information
Platform (PaaS) Business Support Development
Infrastructure (IaaS) Server Database Storage Systems
Cloud Computing
Mobile Computing
Wireless Access Point
Radio Antenna/Transmission Tower
21
3G/WIFI/GPRS
Mobile Devices

Figure 1. Mobile cloud computing architecture [3].

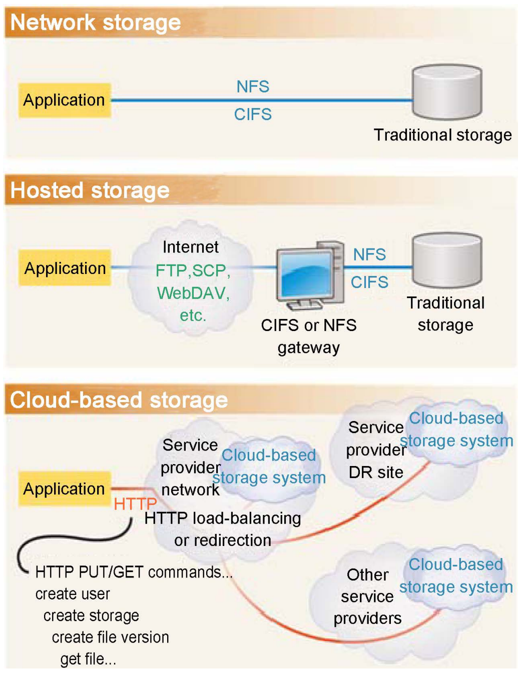

Figure 2. Evolution of cloud storage [4].

store data either using the Network Files System or the Common Internet File System directly or via the Internet. Hosted storage employs clouds owned by cloud service providers to enable applications to store users' data via the Internet.

Cloud storage consists of logical pools that physically span to multiple servers and possible locations. These physical environments are usually owned and managed by cloud companies (called hosting cloud storage providers) such as Microsoft Azure and Amazon Web Services.

The main responsibilities of these cloud storage providers are to maintain the data and ensure it is available and accessible to its clients, as well as to protect the physical environment and ensure it remains running efficiently. Organizations and people lease or buy storage capacity from these cloud providers to store data for users, organizations, and/or applications [5].

Cloud storage systems occur in many different varieties, *i.e.*, several storage systems are specialized and have a specific focus (e.g., the storage of web email messages or the storage of digital pictures), while other cloud storage systems are utilized to store digital data of all forms. Certain cloud storage systems might be described as small operations, while others are so large that their physical equipment can fill up an entire warehouse. The combined facilities that house and compose the cloud storage systems are known as data centers.

Each cloud storage system requires one data server and a connection to the Internet via an ISP. Any client (e.g., mobile user subscribing to any cloud storage service) sends their file(s) over the Internet to the cloud data server and then the cloud data server records the sent information. If the client requested action is to retrieve the information, he/she accesses the corresponding cloud data server via an interface that is web-based. The cloud server can then either send the requested file(s) back to the client who made the request or allow the client to manipulate and access the file(s) on the server itself.

Synchronization is a method to synchronize a single set of data (such as a user's contact list) by automatically copying any changes back and forth between the mobile device and the cloud storage. This process ensures that the user's data can be easily viewed from anywhere by any authorized mobile device. For example, mobile users of Google Drive and Dropbox can access and share their own data in mobile environments. In such applications, a user can add or modify a file in his/her local folder and this update is then automatically reflected (synchronized) by a cloud server. In the case of sharing a file with other users, the cloud server takes the responsibility of synchronizing this shared file with all sharing users. If this file has frequent updates, it generates a significant amount of traffic during the synchronization process, which leads to challenges with trying to determine ways of reducing the synchronization traffic and its required time. These challenges constitute one of the most important issues in applications that employ data sharing [6].

A major issue in MCC synchronization is how to guarantee the data that is required to be synchronized given the different uncertainties, e.g., disconnections before process completion and insufficient cloud storage space. The present study addresses the problems of synchronization, discusses a synchronization method utilized by one major cloud service provider, and proposes a new method of synchronization.

This work proposes an algorithm, based on Delta synchronization, to solve the problem of synchronizing compressed files under various forms of modification (e.g., not modified, partially modified, or completely modified). To measure the efficiency of our proposed algorithm, we compared it to the Dropbox application algorithm.

This paper is organized as follows. Section 2 covers a literature review related to the research, while section 3 explains the concept of Delta synchronization. In section 4, the research problem is defined, and a theoretical solution and our proposed algorithm design are described. The experimental setup is shown in section 5, while results are discussed in section 6. Section 7 provides the conclusions of the study.

2. Literature Review

Many mobile cloud applications share data among various users and synchronize with their corresponding cloud servers. In [7], an Update-triggered Delta Synchronization (UDS) algorithm was proposed, which is a popular solution for data synchronization in mobile cloud applications.

Delta encoding computes the difference (*i.e.*, Delta) between the old and new versions of a file. When a shared file is modified, only the difference in the file computed by the algorithm is notified to the users and the cloud server for synchronization. If the shared file is frequently modified, and a large number of users are sharing the file, the proportion of synchronization traffic of the UDS algorithm is deemed significant. Therefore, a proxy-based aggregated synchronization scheme has been proposed to reduce the synchronization overhead when frequent updates were applied to a shared file being accessed (shared) by many users [8].

An efficient Delta synchronization (EDS) algorithm that improves the performance of the UDS has been proposed [9]. The algorithm aggregates updated data (*i.e.*, Delta) in a step aimed at reducing the network traffic, and then synchronization is applied to the aggregated updates in a periodic manner to reflect modifications and satisfy consistency of files. In MCC, the cloud service is utilized to resolve the resource constraint problem of mobile device offloading, which causes excessive mobile battery consumption.

In Min & Hei work [10], a response time-based model was proposed, where data synchronization was perceived to estimate the efficiency of the offloading scheme in terms of response time. The goal was to improve the accuracy of the response time estimation when the cloud processes the requested task received from any mobile device.

Domingos *et al.* [11] proposed adapting the Synchronization Algorithms based Message Digest to the mobile environment to minimize the usage of mobile device resources (*i.e.*, reducing the amount of data transmitted during the synchronization process and reducing the processing carried out by the mobile device) by using only standard Structure Query Language (SQL) queries. The major goal for this proposed model was to minimize the amount of data exchange.

Furthermore, the authors of [12] proposed a model to Improve Synchronization Algorithms based on Message Digest (ISAMD) to resolve inconsistencies between the server-side database and the mobile client database.

Asynchronous data synchronization model that blocks the user interface during synchronization to prohibit any updates during the synchronization process has also been proposed [13], although this model disables other processes during synchronization and is disliked by the majority of users.

3. Delta Synchronization

Delta synchronization is one of the available designs that can assist in solving the challenges facing synchronization [14]. It is a new feature that increases the speed of transfer of updated files by only uploading/downloading the modified parts of the files, instead of sending/receiving the entire files whenever the files are changed.

3.1. Description

Figure 3 provides an example of Delta synchronization, showing six users on the left hand side, with two of these users (numbers 2 and 6) altering their data (different colors indicate a change) and the remaining users (1, 3, 4, and 5) not altering their data. For the latter users who have not changed their data, we avoid re-synchronizing the data and synchronization was only applied to the data of users 2 and 6 as their data had changed.

3.2. Solution

A variant of Delta Synchronization called EDS is shown in **Figure 4**.

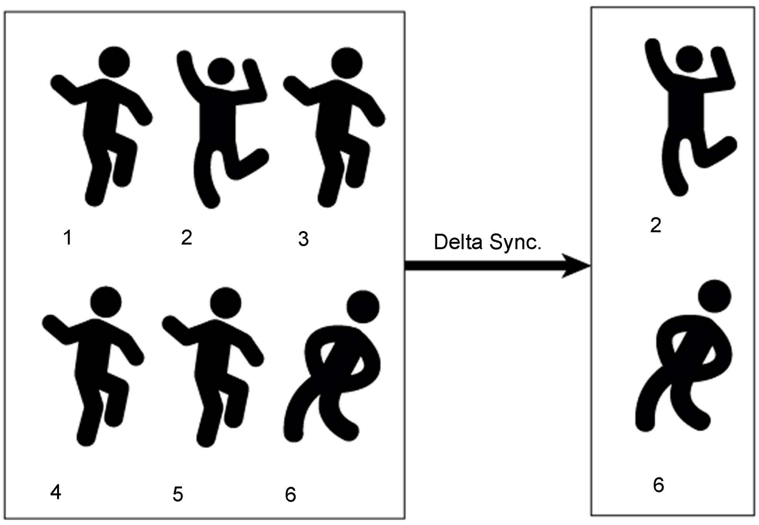

Figure 3. Delta synchronization example.

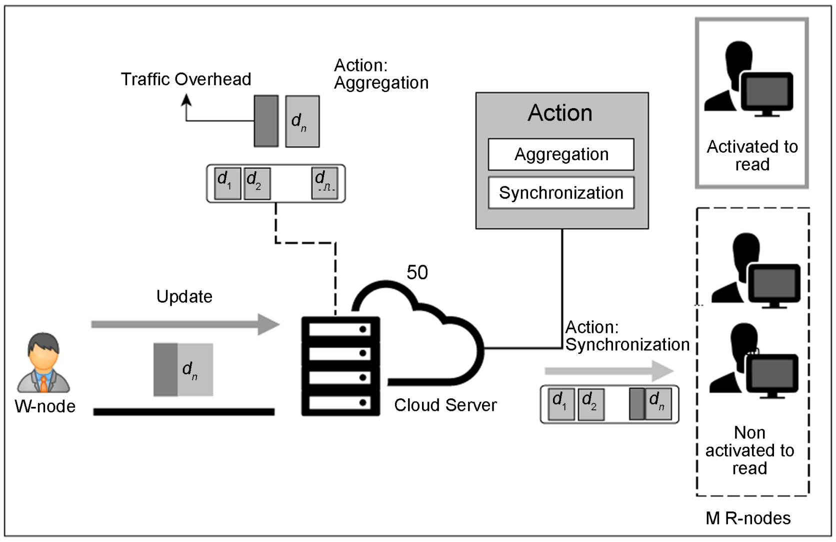

Figure 4. Efficient delta synchronization [9].

In EDS, a given file F is shared by N+1 mobile nodes and the file is synchronized with the corresponding cloud server. Let F′be a newer version of F and d denotes the difference between F and F′. The nth update of d is denoted by d_n. In this environment, a mobile node can be one of the following two types: a write node (*i.e.*, W-node) or a read-only node (*i.e.*, R-node). To avoid simultaneously updating the F file from more than one node, only one node is allocated as a W-node while the remaining M nodes are allocated as R-nodes. Regardless of whether the file is compressed or not, the W-node is able to update d of the file F onto the cloud server. During the writing connection, the update process requires a T time interval. At the end of the writing connection of the W-node (*i.e.*, once the process is terminated) another new W-node is selected. On the other side of this process, the file F can be read by activating the R-nodes during the T time interval. Any activated R-node during the read-only operation of the file F is called an activated R-node.

The synchronization process commonly involves extra overhead owing to the setup and maintenance of the TCP/HTTP(S) and the delivery of metadata between the cloud server and the R-node. Aggregating the updates from the W-node in the cloud server and propagating them to all the R-nodes to complete synchronization later greatly assists in mitigating the traffic overhead. As a result, a d update from the W-node is determined to be either aggregated in the cloud server or synchronized with all R-nodes at each T time interval. Although update aggregation can mitigate the traffic of synchronization, it can also cause inconsistency between the activated R-nodes and the W-node; in such case and when the updates are not yet modified, the activated R-node could possibly use an F file that is outdated.

3.3. Motivation

Even though Delta synchronization assists in solving the problem of redundant data synchronization that decreases the cloud storage usage, it is problematic when dealing with new mobile device users who frequently access large files. Generally, large files (e.g., video, music, digital photo, Photoshop files, among others) are stored compressed; therefore, a change in a small part of a compressed file requires re-synchronizing the whole file because Delta synchronization does not deal with file parts. Therefore, there is a need for the proposal of a new model to be developed to overcome this problem.

4. Proposed Method

Delta synchronization is a reasonable solution when handling regular files. However, it significantly fails when handling large compressed files, with the main source of failure being that the whole file has to be re-synchronized instead of merely updating the sub-files. This causes network congestion and reduces the available bandwidth, especially when synchronizing mobile devices in a wireless environment.

One possible solution to the problem of synchronizing large compressed files is to synchronize only the altered sub-files contained within the compressed file. Given that compressed files normally have many sub-files, a logical method to update the files contained within is to decompress the files, synchronize the updated sub-files, and then recompress the files. This process might take a considerable amount of time, especially when there is a large number of compressed files. Therefore, the development of a method to synchronize only the changed sub-files without having to send the whole folder and have to go through the lengthy process of extracting and compressing is required. Such a method would reduce the time, traffic overheads, and synchronizing costs. In the present study, we propose a method (an algorithm) that will only update the sub-files of a large compressed file.

Specifically, the proposed algorithm employs the concepts of hashing or hash functions. Hashing is utilized to convert several data types into a relatively small number that could serve as digital "fingerprint" data. The algorithm used in hashing is the one that manipulates the data to create the fingerprints that are called hash values or hash codes.

Figure 5 illustrates a client who has a database that contains a hash code for each synchronized file and its sub-files (of the compressed file), *i.e.*, a client who has a compressed file and needs to synchronize it. Our synchronizing algorithm works by first calculating the hash code and, based on it, assumes one of three cases: 1) the compressed file is new, 2) the compressed file is not updated, and 3) the compressed file is updated. In the third case, four different scenarios for the sub-files are possible: a) the sub-file is new (*No Update is Needed*), b) the sub-file exists but has not been updated (*Inserting a New Sub-file*), c) the sub-file exists and has been updated (*Updating an Existing Sub-File*), and d) the sub-file has been deleted (*Deleting an Existing Sub-file*). At the end, the algorithm applies the necessary synchronization and deletes the resulting lists. A detailed description of our proposed synchronization algorithm is described in Figure 6.

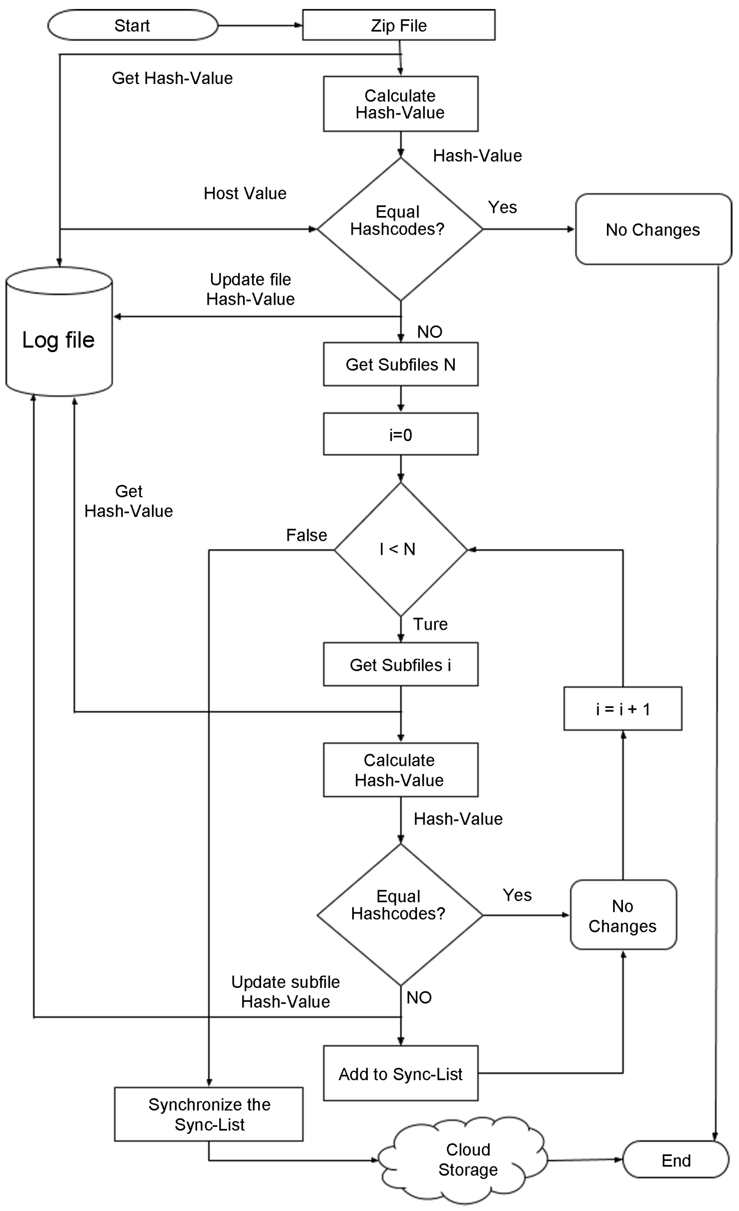

Figure 5. Compression delta synchronization system.

5. Experimental Design

The hardware and software utilized play an important role in the performance of any algorithm subject evaluation or study. For algorithm implementation and experiment design in the present study, a laptop with regular hardware (*i.e.*, Core i5-5200U CPU @ 2.20 GHz, 8 GB RAM) and software (*i.e.*, Windows 10 Pro x64) specifications was utilized. We implemented our proposed algorithm under Visual Studio 2010, with C#. Furthermore, MD5 hashing function was already embedded within the .Net 4.0. Even though MD5 has some security concerns and can be figured by simple reverse engineering technique such as Brute-force attacks, it was selected because of its calculation speed. The disadvantages of using MD5 fall outside the scope of the present study and should be discussed further in the future.

For simplicity, a small database architecture was created. The database contained just one table with three columns (Table 1). The *File Parent* column was utilized to hold the name of the compressed file, while the *File FullName* column was utilized to hold the name of the sub-file(s). Finally, the *File Hash* was utilized to hold the value of the Hash code of *File FullName*.

The main motive behind creating this database was to summarize files saved on the client's device. In addition, changes during the synchronization process were saved as there was no need to monitor each change in all of the client's files.

Step 1: The proposed algorithm calculates the **HashCode** for the compressed file subject to synchronization without calculating the HashCode for the sub-files in the compressed file; the following three cases should be considered:

- ***Case 1:*** *if the* ***compressed file is new*** *(by checking that the file has not been inserted into the database), then the algorithm calculates the HashCode for each sub-file without extracting the compressed file. After that, the algorithm adds them (the sub-files in the new compressed file) to* ***fileToSync*** *list (this list contains the files that should be synchronized) and inserts them into the database.*
- ***Case 2:*** *if the* ***compressed file is not updated*** *(the file exists in the database with the same HashCode), then the file has already been synchronized and the algorithm does nothave to do anything.In this case,the algorithm compares only the HashCode of the* ***CompressedFile****(with the HashCode it already has in the database) without comparing the HashCode for each sub-file.*
- ***Case 3:*** *if the* ***compressed file is updated*** *(the file exists in the database but with a different HashCode, we identify which file was updated by using CompressedFile and* ***FileFullName****(full location path of the file on the server, with file name and extension)), then the algorithm calculates the HashCode for each sub-file without extracting the CompressedFile. In this case, the following four scenarios are possible for each sub-file to determine if the algorithm needs to synchronize that sub-file or not:*
    - ***Scenario 1:The sub-file is new*** *(the sub-file is not inserted in the database), then the algorithm inserts it into the database and adds it to the list of files that need to be synchronized.*
    - ***Scenario 2:The sub-file exists but not updated*** *(the sub-file is available in the database with the same HashCode), then the algorithm does not need to synchronize the sub-file.*
    - ***Scenario 3:The sub-file exists and updated*** *(the sub-file is available in the database but with different HashCode), then the algorithm updates its HashCode in the database and adds it to the list of files that need to be synchronized.*
    - ***Scenario 4:The sub-file is deleted*** *(the sub-file exists in the database butdoes not exist in the CompressedFile), then the algorithm deletes it from the database and adds it to the* ***fileToDelete*** *list (the list of files that do not exist in the CompressedFile and should be deleted from the database) that need to be deleted.*

Step 2:*At the end, and to prepare for the next synchronization round, the algorithm applies the* ***necessary****synchronization and deletes the resulting lists (****fileToSync****,* ***fileToDelete****) and gets ready for another round.*

Figure 6. Proposed synchronization algorithm.

Table 1. Database architecture: files table.

File Parent	Text
File FullName	Text
File Hash	Text

6. Results and Discussion

To analyze the results of our proposed algorithm, we conducted two different tests (Test 1 and Test 2). In Test 1, the time required to perform an operation was recorded and compared between various file sizes, while in Test 2 the proposed method was benchmarked in a live environment against a popular cloud service provider. Results of the Test 1 experiment are reported in Table 2, which summarizes the results we recorded running our proposed algorithm on a number of different compressed files, each of which had a different file size and file type (e.g., text, image, audio, and video). For each compressed file, five different synchronization times for five different cases were recorded. First, the required time needed for inserting a new compressed file was recorded. Then, the time necessary to determine that there was no need to update was calculated, *i.e.*, the time required for calculating and comparing the hash codes, the time needed when a new sub-file had to be inserted, the time needed for updating that file, and the time needed to delete a sub-file. Regarding the state where *No Update is Needed* involved synchronizing the same file without any changes. It is shown in Table 2 that our proposed algorithm performed very well as it did not upload any file and took a relatively short time to ensure that no changes were available while calculating the HashCode. *Inserting a New Sub-file* was concerned with the state where a new file had been inserted into the compressed file. In our proposed algorithm, only the new sub-file was synchronized instead of updating the whole compressed file. *Updating an Existing Sub-File* case dealt with updating one of the sub-files in the compressed file; therefore, the updated sub-file was synchronized instead of synchronizing the whole compressed file. Finally, *Deleting an Existing Sub-file* was the state when the existing sub-file available inside the compressed file had been deleted. Our proposed algorithm was capable of deleting the targeted sub-file from the compressed file, and there was no need to synchronize the whole compressed file.

The result revealed that the time for calculating the HashCode (no update) was consistently the minimum for each of the nine files (Table 2, column 3) and the maximum time needed was for inserting a new compressed file (Table 2,

Table 2. Test 1: compressed files recorded results.

File size (MB)	Insert new file (ms)	No update (ms)	Insert new sub-file (ms)	Update sub-file (ms)	Delete sub-file (ms)
36	1423	320	681	653	638
45	5313	467	1,231	874	859
64	4783	623	4629	4577	4536
105	7217	922	6671	6649	6725
123	6006	1093	3417	3372	3385
137	9791	970	3806	3837	3695
213	5363	1023	3625	2813	2843
664	53,225	7000	36,537	36,483	36,998
1710	>65,800	>13,000	>79,000	>81,000	>80,000

column 2). As shown in Table 2 and by comparing the recorded times for all cases, it can be concluded that our proposed algorithm significantly reduced the time needed to perform a specific operation (e.g., insert, update, delete, and so on) resulting in a reduced cost. However, the sub-files that were subject to modifications in all cases were mainly small files; hence, the experiment was subject to file type/size adjustments.

To benchmark the efficiency of our algorithm, we compared our results with a popular live cloud synchronizing environment, the Dropbox application (Dropbox, Inc., headquartered in San Francisco, California, USA, [15]). In this test, called Test 2, Case 1, we developed an application to upload a file to Dropbox while measuring the time needed for uploading the file. Table 3 summarizes the performed experiments. In brief, five different groups, each of which contained three files of the same type (e.g., text, image, audio, video, and mixed type) with various file sizes were utilized.

As shown in Table 3, the original file size before and after compression was calculated before performing other operations. For file compression, we applied the free WinZip tool embedded within Windows. Further, the compression ratio (CR) was calculated and then reported as shown in Equation (1):

$$CR = OFS/CFS, \qquad (1)$$

where, OFS denotes the original file size and CFS represents the compressed file size.

Table 3 reveals that the compression ratio for text files was higher than for other file types, and reports the time required to insert the compressed file completely, while the other reported results are similar operations to those explained for Table 2.

Table 3. Test 2, case 1: Dropbox performance.

File Type	Original file size (MB)	Compressed file size (MB)	Compressed ratio (%)	Insert new compressed file (ms)	No update (ms)	Insert new sub-file (ms)	Update sub-file (ms)	Delete sub-file (ms)
1) Text	22.866	0.368	62.14	5547	4620	5865	5819	5973
2) Text	91.464	1.471	62.18	18,082	16,384	18,907	17,171	15,964
3) Text	548.789	8.824	62.19	96,671	91,745	95,875	90,451	89,881
4) Image	6.453	5.969	1.08	73,150	62,517	65,426	65,161	63,401
5) Image	12.757	11.793	1.08	144,400	126,933	136,131	127,689	125,989
6) Image	16.856	15.623	1.08	170,872	165,042	168,775	167,931	164,182
7) Audio	0.789	0.77	1.02	10,639	8692	16,339	11,940	9001
8) Audio	8.21	8.197	1.00	89,009	87,143	102,964	103,520	87,446
9) Audio	11.801	10.961	1.08	118,074	114,280	115,088	116,505	112,339
10) Video	9.157	9.126	1.00	113,512	106,765	111,954	112,978	105,092
11) Video	11.272	11.243	1.00	122,445	121,639	123,645	135,750	119,542
12) Video	18.313	18.289	1.00	196,396	191,153	200,863	209,934	190,158
13) Mixed	7.81	7.62	1.02	95,149	89,694	93,207	93,805	90,117
14) Mixed	15.421	7.981	1.93	100,791	88,043	107,824	103,753	99,635
15) Mixed	33.675	12.366	2.72	129,351	127,173	136,935	139,762	128,964

It can be seen that the Dropbox application did not reduce the synchronization time when any change occurred during the synchronization process for all file sizes and types as the whole compressed file had to be re-synchronized. The slight change in time between the different cases (e.g., Table 3 columns 5 through 9) were expected and could be attributed to the load on the Internet or the bandwidth of the network hosting the mobile.

We applied our proposed algorithm to the same dataset used in the Dropbox experiment (Test 2, Case 1) and recorded the results in Table 4; we called this experiment Test 2, Case 2.

In Table 4, it can be seen that the time needed to perform the same operation was reduced by a great magnitude when applying our proposed algorithm compared to the time required by the Dropbox application reported in Table 3.

The changes we applied to the sub-files for each row of Table 3 and Table 4 are recorded in Table 5. These changes were applied for Case 1 and Case 2 of Test 2.

Table 6 contains the performance gain achieved by our algorithm and shows only the performance gain percentage (*PGP*) for clear and useful comparison. We calculated the *PGP* as shown in Equation (2):

$$PGP = \frac{(DbxT - PrT)}{DbxT} \times 100\%, \tag{2}$$

where $DbxT$ is the time required by the Dropbox application and PrT is the time required by our proposed algorithm.

Table 6 shows the *PGP* we achieved by applying our proposed algorithm to the different cases. When no update occurred at all, the performance gain was slightly higher than 99.5% while the lowest reported performance gain was for the text files. On average, the performance gain for inserting the compressed text

Table 4. Test 2, case 2: proposed algorithm performance.

File Type	Original file size (MB)	Compressed file size (MB)	Compressed ratio (%)	Insert new compressed file (ms)	No update (ms)	Insert new sub-file (ms)	Update sub-file (ms)	Delete sub-file (ms)
1) Text	22.866	0.368	62.14	2955	206	874	737	658
2) Text	91.464	1.471	62.18	6066	51	1891	1816	1735
3) Text	548.789	8.824	62.19	36,780	84	9150	9113	9058
4) Image	6.453	5.969	1.08	2500	72	1163	1064	983
5) Image	12.757	11.793	1.08	4769	96	3126	1689	1686
6) Image	16.856	15.623	1.08	6066	112	2638	2956	2304
7) Audio	0.789	0.77	1.02	332	53	339	569	352
8) Audio	8.21	8.197	1.00	1324	82	910	1188	699
9) Audio	11.801	10.961	1.08	1439	92	865	932	838
10) Video	9.157	9.126	1.00	634	86	521	483	421
11) Video	11.272	11.243	1.00	853	94	573	697	517
12) Video	18.313	18.289	1.00	1180	145	961	956	679
13) Mixed	7.81	7.62	1.02	645	74	473	454	384
14) Mixed	15.421	7.981	1.93	1516	82	747	985	668
15) Mixed	33.675	12.366	2.72	2732	99	1489	1559	1117

Table 5. Changes applied to all sub-files.

File Type	Sub-file size	Update size
1) Text	0.141 MB	Adding 5 Characters
2) Text	0.846 MB	Adding 20 Characters
3) Text	1.691 MB	Deleting 10 Characters
4) Image	0.126 MB	Rotating image
5) Image	0.149 MB	Rotating and filtering image
6) Image	1.437 MB	Rotating 3 images and changing the light of 1 image
7) Audio	0.630 MB	Minimizing the same file to 0.212 MB
8) Audio	1.510 MB	Enlarging the same file to 1.720 MB
9) Audio	0.186 MB	Enlarging the same file to 0.210 MB
10) Video	0.850 MB	Enlarging the same file to 0.950 MB
11) Video	0.953 MB	Enlarging the same file to 2.139 MB
12) Video	0.815 MB	Minimizing the same file to 0.437 MB
13) Mixed	Text file 0.141 MB and an audio file 0.815 MB	Adding 5 Characters to the new text file and rotating the new image
14) Mixed	Text file 0.216 MB and a video file 1.187 MB	Adding 10 Characters to the new text file and minimizing video to 0.753 MB
15) Mixed	Text file 0.216 MB and image file 0.137 MB and an audio file 0.815 MB and a video file 1.187 MB	Adding 5 Characters to the new text file and rotating the new image and maximizing video to 1.283 MB

Table 6. Percentages of performance gain improvement.

File Type	Original file size (MB)	Compressed file size (MB)	Compressed ratio (%)	Insert new compressed file (%)	No update (%)	Insert new sub-file (%)	Update sub-file (%)	Delete sub-file (%)
1) Text	22.866	0.368	62.14	46.728	95.541	85.098	87.335	88.984
2) Text	91.464	1.471	62.18	66.453	99.689	89.998	89.424	89.132
3) Text	548.789	8.824	62.19	61.953	99.908	90.456	89.925	89.922
	Average		**62.17**	**58.378**	**98.379**	**88.518**	**88.895**	**89.346**
4) Image	6.453	5.969	1.08	96.582	99.885	98.222	98.367	98.450
5) Image	12.757	11.793	1.08	96.697	99.924	97.704	98.677	98.662
6) Image	16.856	15.623	1.08	96.450	99.932	98.437	98.240	98.597
	Average		**1.08**	**96.577**	**99.914**	**98.121**	**98.428**	**98.569**
7) Audio	0.789	0.77	1.02	96.879	99.390	97.925	95.235	96.089
8) Audio	8.21	8.197	1.00	98.513	99.906	99.116	98.852	99.201
9) Audio	11.801	10.961	1.08	98.781	99.919	99.248	99.200	99.254
	Average		**1.03**	**98.058**	**99.739**	**98.763**	**97.762**	**98.181**
10) Video	9.157	9.126	1.00	99.441	99.919	99.535	99.572	99.599
11) Video	11.272	11.243	1.00	99.303	99.923	99.537	99.487	99.568
12) Video	18.313	18.289	1.00	99.399	99.924	99.522	99.545	99.643
	Average		**1.00**	**99.381**	**99.922**	**99.531**	**99.535**	**99.603**
13) Mixed	7.81	7.62	1.02	99.322	99.917	99.493	99.516	99.574
14) Mixed	15.421	7.981	1.93	98.496	99.907	99.307	99.051	99.330
15) Mixed	33.675	12.366	2.72	97.888	99.922	98.913	98.885	99.134
	Average		**1.89**	**98.569**	**99.916**	**99.237**	**99.150**	**99.346**
	rand Average		**13.436**	**90.192**	**99.574**	**96.834**	**96.754**	**97.009**

files was only 58%. Even though this value was lower than that for the other file types, it still remains a significantly high performance gain. The performance gain for inserting a compressed file was greater than 96% for all other file types. A possible reason for the relatively low performance gain reported for text files might be the high compression ratio compared to other file types. However, the problem of inspecting the details of this variability is not in the scope of this paper and requires future research.

7. Conclusion and Future Work

In the present study, we proposed a new algorithm based on Delta synchronization that greatly improved the synchronization process and solved the problem of synchronizing compressed files. Two tests to validate the proposed algorithm were designed that compared the required time for achieving only specific changes in the compressed file instead of updating the whole file, and that benchmarked our proposed algorithm by comparing its performance to the Dropbox application that synchronizes the whole file after any amendment. Our proposed algorithm proved to work more efficiently than the Dropbox application did and demonstrated great potential in the field of cloud synchronization. However, there is stillroom for improvement, with testing of different scenarios and benchmarking with various providers required.

References

[1] Kaur, K., Sharma, S. and Arora, M. (2014) Mobile Cloud Computing Techniques: A Review. *International Journal of Advanced Research in Computer Engineering & Technology*, **3**, 4.

[2] Dinh, H.T., Lee, C., Niyato, D. and Wang, P. (2013) A Survey of Mobile Cloud Computing: Architecture, Applications, and Approaches. *Wireless Communications and Mobile Computing*, **13**, 1587-1611. https://doi.org/10.1002/wcm.1203

[3] Asrani, P. (2013) Mobile Cloud Computing. *International Journal of Engineering and Advanced Technology*, **2**, 606-609.

[4] Rajan, A.P. (2013) Evolution of Cloud Storage as Cloud Computing Infrastructure Service.

[5] Wu, J., Ping, L., Ge, X., Wang, Y. and Fu, J. (2010) Cloud Storage as the Infrastructure of Cloud Computing. *International Conference on Intelligent Computing and Cognitive Informatics*, Kuala Lumpur, 22-23 June 2010, 380-383.

[6] Kaur Saggi, M. and Singh Bhatia, A. (2015) A Review on Mobile Cloud Computing: Issues, Challenges and Solutions. *International Journal of Advanced Research in Computer and Communication Engineering*, **4**, 29-34.

[7] Li, Z., Wilson, C., Jiang, Z., Liu, Y., Zhao, B.Y., Jin, C., Zhang, Z. and Dai, Y. (2013) Efficient Batched Synchronization in Dropbox-Like Cloud Storage Services. *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, Beijing, 9-13 December 2013, 307-327. https://doi.org/10.1007/978-3-642-45065-5_16

[8] Lee, G., Ko, H. and Pack, S. (2014) Proxy-Based Aggregated Synchronization Scheme in Mobile Cloud Computing. *IEEE Conference on Computer Communications Workshops*, Toronto, 27 April-2 May 2014, 187-188. https://doi.org/10.1109/infcomw.2014.6849218

[9] Lee, G., Ko, H., Pack, S. and Kang, C.H. (2014) Efficient Delta Synchronization Algorithm in Mobile Cloud Networks. 3*rd International Conference on Cloud Networking*, Luxembourg, 8-10 October 2014, 455-460. https://doi.org/10.1109/cloudnet.2014.6969037

[10] Min, H. and Heo, J. (2015) Response Time Analysis Considering Sensing Data Synchronization in Mobile Cloud Applications. *The Journal of the Institute of Internet, Broadcasting and Communication*, **15**, 137-141. https://doi.org/10.7236/JIIBC.2015.15.3.137

[11] Domingos, J., Simões, N., Pereira, P., Silva, C. and Marcelino, L. (2014) Database Synchronization Model for Mobile Devices. 9*th Iberian Conference on Information Systems and Technologies*, Barcelona, 18-21 June 2014, 1-7.

[12] Balakumar, V. and Sakthidevi, I. (2012) An Efficient Database Synchronization Algorithm for Mobile Devices Based on Secured Message Digest. *International Conference on Computing, Electronics and Electrical Technologies*, Nagercoil, 21-22 March 2012, 937-942.

[13] McCormick, Z. and Schmidt, D.C. (2012) Data Synchronization Patterns in Mobile Application Design. *Proceedings of the* 19*th Conference on Pattern Languages of Programs*, Tuscon, 19-21 October 2012, 12.

[14] ktacket (2015) The Complete Synchronization Process—Part 4: Delta/Full Import/Synchronization Explained. https://blogs.msdn.microsoft.com/connector_space/2015/09/28/the-complete-synchronization-process-part-4-deltafull-importsynchronization-explained/

[15] Dropbox, Inc., Headquartered in San Francisco, California, USA.