Received August 21, 2020, accepted September 2, 2020, date of publication September 8, 2020,
date of current version September 22, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3022463

Data Consistency in Multi-Cloud Storage Systems With Passive Servers and Non-Communicating Clients

NARAM MHAISEN AND QUTAIBAH M. MALLUHI, (Member, IEEE)
Department of Computer Science and Engineering, Qatar University, Doha 2713, Qatar

Corresponding author: Naram Mhaisen (naram@qu.edu.qa)

This work was supported by the Qatar National Research Fund (Member of the Qatar Foundation) under Grant NPRP 8-2158-1-423.
Open Access funding provided by the Qatar National Library.

ABSTRACT Multi-cloud storage systems are becoming more popular due to the ever-expanding amount of consumer data. This growth is accompanied by increasing concerns regarding security, privacy, and reliability of cloud storage solutions. Ensuring data consistency in such systems is especially challenging due to their architecture and characteristics. Specifically, passive cloud storage cannot run coordination software, and clients cannot communicate directly to reach consensus. Furthermore, the atomicity of operations is not always guaranteed by the clouds' public APIs. In this paper, we formally define data consistency in multi-cloud storage systems, identify how they can be violated, and introduce a new method that provably maintains the data consistency in these systems. The implementation and experiments show that the proposed method can maintain data consistency with a certain delay in data uploading and that it is scalable with respect to the number of used clouds as well as the number of users. Integrating this method into multi-cloud storage systems will enhance their usability and reliability.

INDEX TERMS Cloud-of-clouds, cloud privacy, cloud reliability, data consistency, multi-cloud storage.

I. INTRODUCTION

With the ever-growing amounts of data, users are shifting towards cloud storage services. These services provide convenience as they omit the need for maintaining local storage means and provide accessibility from anywhere and across different devices [1], [2]. However, outsourcing data to the cloud comes with data confidentiality and integrity critical requirements [3], [4]. Relying on a single cloud storage provider fails to meet these requirements due to the inevitable risks of privacy breaches, data leaks, and service outages [5].

To tackle this issue, various multi-cloud storage systems (also known as a cloud of clouds) have been proposed in the literature. Most of these systems partition the data into multiple parts, encode these parts using erasure codes, encrypt them, and finally, save each part on a different cloud provider [6]. When such systems are server-less (i.e., partitioning, encoding, and encryption operations occur on clients' machines rather than on a centralized server), they can offer privacy, security, and protection from data loss. Privacy is guaranteed since individual cloud providers will have no knowledge about the content of the data as they store only an encrypted part. The data is also secured since its integrity is preserved (since modifications will lead to detectable data corruption). Finally, reliability is provided through erasure codes since even if part of the data is unavailable (e.g., due to an inaccessible cloud), the original data can still be reconstructed/decoded from other available parts. In general, server-less multi-cloud storage systems can provide trust in the cloud.

There are several important use-cases of multi-cloud-storage systems for both end-users as well as businesses. End users can store personal files that are hidden from the cloud and not subject to potential denial of access. This is becoming more important due to multiple recently reported privacy-leak incidents, which caused many cloud-end users to opt-out of using cloud services [7]. Similarly, multiple businesses would also like to have guaranteed privacy and availability of their sensitive data. This is especially important for businesses since they are subject to different jurisdictions and potential subpoena-forced data acquisition or service denial, depending on the cloud data center location [8].

The associate editor coordinating the review of this manuscript and approving it for publication was M. Shamim Hossain.

Multi-cloud storage systems, similar to cloud storage services, should allow users to access and modify their files from anywhere. Furthermore, users can access their data from multiple independent client devices. Therefore, data should always be synchronized and consistent across all users' devices. One of the fundamental synchronization features is the ability to detect data conflicts and maintain data consistency [9]. In general, data conflicts occur when multiple clients attempt to modify the same file at the same time. Data consistency assures that no information is lost in such a case.

Multi-cloud storage systems can detect conflicts and preserve consistency through utilizing a centralized coordination point (e.g., server) that receives and logs the modification requests from the different clients (append-log). A specialized software can parse the logs and determines the existence of a conflict. However, secure multi-cloud storage systems are server-less. Hence, there is no central controller to coordinate between clients and detect data conflicts. This is of utmost importance since users should not need to trust any third party to handle their raw data.

The more popular multi-cloud storage system design uses a distributed system. In such a case, the well-established consensus algorithms, such as Paxos [10] and its variants, are utilized. However, the characteristics of multi-cloud storage systems represent a specific situation that violates many assumptions of these consensus methods. Specifically, cloud storage clients cannot effectively communicate and coordinate with each other to execute the consensus protocol since they connect to the service only when they need. In fact, they do not know each other and, therefore, cannot establish peer-to-peer communication. Hence, the lack of communication between clients is a significant challenge to the consensus-based approach. Additionally, for basic cloud storage services, there are no processing resources offered by cloud storage providers. The cloud providers only support write/read and related operations (passive servers). Thus, programs and protocols cannot be executed on these servers.

An additional and essential challenge that faces multi-cloud storage systems is the heterogeneity of consistency models followed by different providers [11]. Having a strict consistency assumption or atomicity of operations from a cloud storage provider is an impractical assumption that should be avoided. Current cloud storage providers are mostly adopting the *eventual consistency* model. Under such a model, any read/write sequence results cannot always be guaranteed to return the same results [12]. Nonetheless, a reliable multi-cloud storage system should provide an application-level mechanism that ensures data consistency despite the lack of atomic operations or consistency guarantees at the individual cloud level.

To summarize, conflict detection in multi-cloud storage systems is unique and challenging for the following reasons:

- Client-side only.
- Lack of communication between clients
- Lack of processing resources on passive storage servers
- Multiple independent destinations with different consistency models, potentially leading to conflict detection in some clouds but not others
- Lack of atomicity of some cloud storage API functions.

In this paper, we investigate multi-cloud storage systems and propose an application-level client-centric consistency method that provably detects data conflicts and resolves them. Such a consistency feature will enhance the usability of multi-cloud storage systems and hence contribute to the establishment of private, secure, and reliable storage to end-users. The contributions of this paper are summarized as follows:

- Formally defining the data consistency problem in the context of multi-cloud storage systems
- Proposing a novel method that guarantees eventual data consistency in multi-cloud storage systems with passive servers and non-communicating clients
- Implementing a multi-cloud storage system that utilizes the proposed method to demonstrate its performance empirically.

The paper is organized as follows: Section II discusses related solutions and addresses their shortcomings. Section III presents the multi-cloud storage system model. We formulate the data consistency problem in multi-cloud storage systems in section IV. Then, we introduce the proposed solution in section V and evaluate its performance in section VI, before concluding in section VII.

II. RELATED WORK

Various multi-cloud systems have been proposed in the literature. Cloud-RAID [13], NCCloud [14], RAIN [15], [16], RAIC [17], Hyprid CoC [18], and Uni4Cloud [19] systems leverage multiple clouds to address the aforementioned cloud trust issue. These systems do not support multiple clients and end-devices and are not prone to concurrent access issues. Thus, the data consistency issue is not considered. Other works like Spystorage [20] and Trustydrive [21] support multiclient access. However, the data consistency issue is either relayed to separate centralized service or is not addressed.

Hybris [22] is a multi-cloud storage protocol. It supports multiple writers consistency. However, inter-client and cloud communication is necessary. This required an extra layer to perform such communication, which is Apache ZooKeeper (ZK). Depending on the configuration, ZK might form a single point of failure since all clients rely on this server to communicate and coordinate updates among each other. Besides, such architecture still requires clients to trust a third party (Apache ZK) while using a multi-cloud storage system.

SLA [23] provides two tree and token-based distributed mutual exclusion algorithms that can be used for multi-cloud storage, but it also requires inter-client communication. MetaSync [24], [25] provides a file synchronization service on top of cloud storage providers. It employs a modified version of the Paxos consensus algorithm called passive Paxos (pPaxos). This modified version allows clients to

communicate passively (through files) over the clouds in order to reach consensus. This modification requires an append-only atomic list to keep track of protocol messages and eventually reach a consensus. Cloud-types [26] proposed specialized data types that guarantee eventual consistency to all clients. A program that utilizes these data types is abstracted from synchronization complexities and can automatically synchronize data through fork-join techniques. The implementation requires communication between servers. Thus, such implementation is not suitable for a multi-cloud storage system where servers are completely passive.

Saveme [27], [28] is a multi-cloud storage system that proposed a mutual exclusion method for concurrent data access without the need for a central server or any communication between clouds or between clients. This method requires atomic operations that cannot be interrupted. The authors surveyed different APIs from several cloud providers to identify some atomic operations and mapped the unlock/lock operations to other atomic operations offered by the cloud providers. While the proposed system successfully addresses the concurrent access issue, it might be unreliable since these operations are not guaranteed to stay atomic by the cloud providers. Also, the implementation is more complicated since each provider offers different atomic operations. For example, placing a lock might be mapped to moving a file in cloud A, whereas in cloud B, it might be mapped to adding a comment to a file. The deadlock recovery method is based on a fixed amount of time that is experimentally determined (deadline), which is not suitable for all network connections and speeds. Lastly, the method selects a cloud out of the used ones to be used for locking/unlocking operations. However, the selected cloud might not be available to all users at the same time. In such cases, the mutual exclusion will not be sufficient.

TABLE 1. Features of different data consistency approaches in multi-cloud storage systems.

Reference	Consistency method	Assumptions about the clouds
[20], [21], [22]	Log parsing	Central coordination point
[23], [26]	Active consensus	Inter client and/or inter server communication
[24]	Passive consensus	Atomic operations
[27], [28]	Algorithmic	Atomic operations
This work	Algorithmic	-

Table 1 summarizes the main approaches to achieve the data consistency feature in multi-cloud storage systems that provide such a feature. The log-parsing approach assumes a central entity that can receive and coordinate between all clients to guarantee conflict detection and resolution. Such an entity constitutes a single point of failure that jeopardizes reliability. This also creates a performance hot spot as all coordination tasks are performed on a single node. On the other hand, in the distributed systems literature, the concept of consensus has been developed and used extensively in cases where distributed entities need to agree on a single value (or plan) and is used in multi-cloud storage systems. However, utilizing consensus protocols (e.g., Paxos and its variants), mandates inter client or inter-server communication (active consensus) [10]. Nonetheless, there has been a line of work that attempts to map the consensus protocol to read/write operations (i.e., passive consensus). For example, the Paxos proposal phase is replaced with file write. Then, based on reading the written files, a proposal can be accepted or rejected. Passive consensus eliminates the need for communication but requires atomic operations to be provided by the cloud API.

This paper builds on the realization that consensus, although sufficient, is not necessary to guarantee data consistency requirements in multi-cloud storage systems as defined later in section V. Instead, it is possible to derive general algorithmic protocols (i.e., not necessarily consensus protocols) that ensure data consistency without strictly requiring consensus between clients. This is especially appealing since it allows some of the necessary assumptions for consensus (active or passive) to be relaxed.

III. SYSTEM MODEL

Fig. 1 shows the architecture of a multi-cloud storage system. Computing systems are networked with multiple cloud storage providers. Cloud consumers store their data files on the storage devices located at multiple cloud storage providers' premises. A cloud consumer's computers use these storage services through cloud access interface functions provided by each cloud provider. The cloud consumer typically has multiple computers sharing and concurrently accessing (reading and writing) the data. A cloud consumer's computers may provide a file access interface within the computer, or to other consumer computers connected to it. In other words,

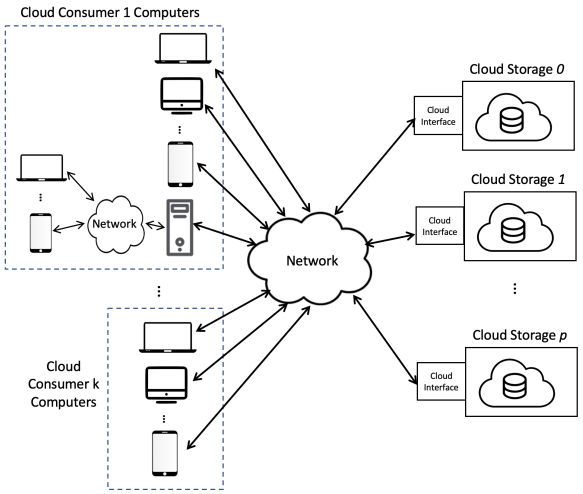


FIGURE 1. Multi-cloud storage system model.

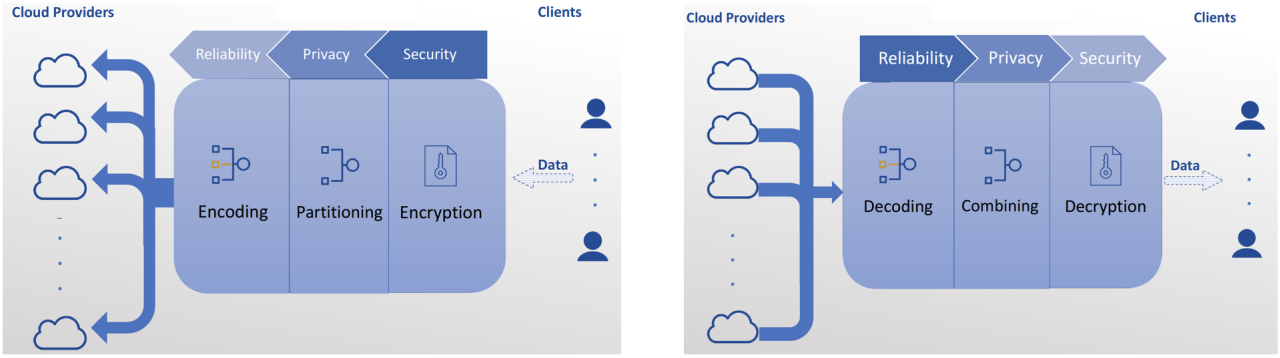


FIGURE 2. Data flow in multi-cloud storage system, writing (left), and reading (right).

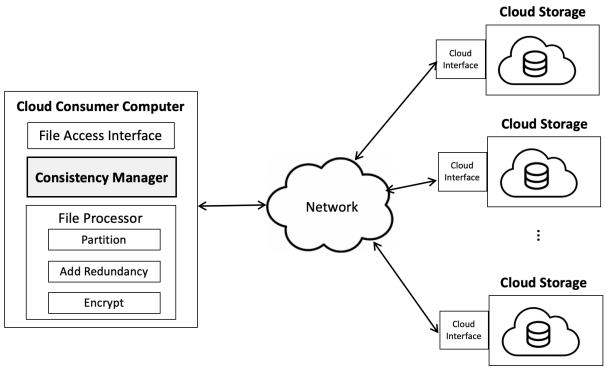


FIGURE 3. Configuration of data consistency management in cloud consumer computers.

some consumers' computers may act as file servers for other computers.

When a file is to be written to the multi-cloud storage system, it goes through multiple stages shown in Fig. 2. The data file is partitioned into n parts. To enhance the availability of data, an optional erasure coding technique can be employed to add redundancy to the data by encoding the n parts into $m = n+t$ parts, such that the original n parts can be retrieved from any n out of the m parts are available. The m parts are encrypted and distributed over m clouds. This is similar to RAID storage systems. For example, RAID5 systems use simple parity to achieve an n out of $n+1$ system that can tolerate the failure of any single storage node. In the general case, an $[n+t, n]$ coding technique is used to achieve a system that can tolerate the failure of up to t storage nodes. In this paper, we design a consistency manager module integrated into multi-cloud clients (see Fig. 3). Note that the file processor module, which implements the data flow pipeline, recognizes parts through their names, which are deterministically derived from the name of the originating file with the addition of a part number. In addition, the encryption key is stored as secret shares saved with the file parts, and produced through an $[n+t, n]$ secret sharing scheme. Therefore, the key can also be reconstructed from the surviving n parts after a potential failure.

IV. PROBLEM FORMULATION

Let D be a data file, and d_n be the n^{th} part of D. We use v_{d_n} to denote the version number of the n^{th} data part d_n, and v_D denotes the version number of the original file D. Let c_n be the n^{th} cloud provider that hosts d_n.

A. CONSISTENCY REQUIREMENTS

Before introducing our method, we formally define data consistency in multi-cloud storage systems:

A multi-cloud uploading algorithm should place data parts d_1,d_2, …,d_n, on the clouds c_1,c_2,…,c_n, respectively, while ensuring that the data parts always satisfy the requirements of a consistent state as specified in the following definition.

Definition-1 (Consistent State): In a multi-cloud storage system, the file D is said to be in a consistent state if and only if:

1) All data parts, d_1,d_2, …,d_n, hosted respectively on the different clouds c_1,c_2,…,c_n, were generated from the same original data D.
2) The versions of all data parts $v_{d_1}, v_{d_2}, \ldots, v_{d_n}$ are written by overwriting a previous versions $v'_{d_1}, v'_{d_2}, \ldots, v'_{d_n}$ (where $v'_{d_i} < v_{d_i}, i = 1, 2, 3, \ldots, n$)
3) All data parts share the same version: $v_{d_1} = v_{d_2} = \ldots = v_{d_n} = v_D$. We refer to parts that share the same version as "homogeneous parts".

For systems that employ an $[n + t, n]$ coding technique, it is sufficient to satisfy the above conditions for n data parts as other parts can always be reconstructed with the same metadata.

To maintain the consistency requirements, it is essential to first identify situations wherein they will be violated; we refer to these situations as the inconsistency cases and identify them in the following subsection.

B. INCONSISTENCY CASES

Definition-2 (Inconsistency Cases): In the multi-cloud storage system, we use the term Inconsistency case to describe situations when:

1) Two or more clients attempt to simultaneously modify the same file (competing to write the same version).

This will violate the first consistency requirement. For example, assuming a file has a certain version v_D, a data conflict occurs when two clients modify this file, each creating version v_D+1 on his/her local machine. Then, they both attempt to write this same version $(v_D + 1)$ on the cloud. Such cases might lead to data parts being heterogeneous (e.g., d_1,d_2 might be coming from client U_1 while d_3 is coming from client U_2).

2) A client attempts to upload a file of version v_D while currently, the file is at another version v'_D where $v'_D \geq v_D$ (overwriting a newer version). This will violate the second consistency requirement. For example, consider the case when a client opens version $v_D = 2$ of a file for read-only, and in the meantime, the file has been modified to version $v'_D = 5$ (three modifications). Then, if that client modifies its already loaded version, a data conflict occurs since he/she will try to overwrite a more recent version with an older one.
3) A client gets interrupted (i.e., stops) at any stage during the upload process. For example, a client might successfully update d_1, but stops before uploading d_2 and d_3. This leads to different versions of the parts, violating the third consistency requirement.

C. DESIGN OBJECTIVES

To prevent the occurrence of all inconsistency cases, a data uploading algorithm has to guarantee the following:

- O_1: Every file being uploaded has a unique uploader (to prevent the first case).
- O_2: An updating upload should be overwriting an older version (to prevent the second case)
- O_3: An updating upload should be interruption-safe. Specifically, if interruption happens while updating a file, a homogeneous set of parts should still be available (to prevent the last case).

To facilitate the discussion, we refer to inconsistency cases 1 and 2 as ''Data conflict cases'' as they arise by concurrent writes or overwriting a newer version, which is already uploaded by another client. We refer to the inconsistency case 3 as an ''Interruption case'' as it is caused by a stopping the upload process by an uploading client unexpectedly.

In the next section, we describe an uploading algorithm that meets these design objectives.

V. PROPOSED DATA CONSISTENCY METHOD

In this section, we propose a method that is able to maintain data consistency requirements without the need for a central server module or/and inter client/cloud communication. The proposed method has two main sections, a cloud-specific section, and a general section that utilizes multiple cloud-specific sections (one for each cloud) to perform the upload process.

The cloud-specific section is detailed in Algorithm 1. It executes for every cloud and deals with the data part hosted on that cloud. This section consists of three main phases. The first phase is placing a temporary file (line 2), which is an indicator that a specific client is uploading a part. The second phase performs two verifications (line 4 for temp file uniqueness, and line 8 for versions). The third phase is temp renaming (which might occur on line 6 or line 10). The output of this algorithm is an indicator of whether a data conflict situation exists on the specific cloud.

Algorithm 1 Cloud Check

Input: *part name*, *part version*
Output: conflict_indicator (True or False)

```
1:  temp= part name + client_ID + ".temp"
2:  Attempt to upload temp
3:    If Failed Return False
4:  Verify that no other temp files for part_file_name exist
5:    If Failed
6:      Remove temp
7:      Return False
8:  Verify that the part version currently on the cloud is older
    than part version
9:    If Failed
10:     Remove temp
11:     Return False
12: Return True
```

Algorithm 2 Upload

Input: *parts*, *file version*
Output: *True* if the original file is updated. *False* if a conflicted copy is uploaded

```
1:  For each cloud c_i, i = 1, ..., n:
2:    c_cloud_indicator = Cloud Check (parts[i], file version)
3:    If c_cloud_indicator is False
4:      parts[i] ← conflicted copy(parts[i])
5:      file version ← v_0
6:      conflict_flag ← True
7:      break
8:  For each cloud c_i, i = 1, ..., n:
9:    Upload parts[i] to parts[i].temp
10:   Upload file version to parts[i].temp metadata
11: For each cloud c_i, i = 1, ..., n:
12:   Rename parts[i].temp to parts[i]
13: Return conflict_flag
```

The general section is illustrated in Algorithm 2. It generally utilizes the cloud-specific section to either permit to write (upload) or not. It is worth mentioning that line 2 can actually be performed by a different thread for each cloud so that all the cloud-specific sections can run simultaneously. Hence, the method represents a multi-threaded network subroutine. Assuming the response time for different clouds is similar, then using more clouds, and therefore more threads, does not affect the execution time as long as the bandwidth can accommodate the parallel requests. This significantly

improves the scalability to more clouds, as will be shown in the performance evaluation section.

If no conflict is detected, the original parts will be uploaded. Otherwise, the client replicates the parts locally, creating another conflicting copy whose parts are to be uploaded. In both cases, each part is uploaded to the corresponding temporary file created earlier by Algorithm 1, referred to as ".temp" (lines 8-10). Then, when all temporary parts are uploaded, the original parts are overwritten by the newer temporary ones (through the renaming in lines 11,12). The returned value is an indicator to the client whether the uploaded copy has successfully overwritten the original file or created a conflicted copy.

A. PROOF OF DATA CONSISTENCY

To prove that the proposed method meets the design objective. We first show that any data conflict situation can be detected. Second, we show that when such detection occurs, then every conflicting client will be the sole uploader of a copy of the conflicted file. Finally, we show that every upload process is interruption safe.

To prove data conflict detection between any number of clients, it is sufficient to prove that the data conflict between any two clients, A and B, can be detected. Furthermore, when any two clients have data conflict, it implies they have a conflict in at least one cloud. Since the general section (Algorithm 2) declares a conflict when any thread of the cloud-specific section (Algorithm 1) returns *False*, it is sufficient to prove that the cloud-specific section can detect the conflict between any two clients.

As per the design of the cloud-specific section of the method, each client will go through three phases before deciding on conflict existence; (1) the temporary file placing phase, (2) the verification phase, and (3) the temporary file renaming phase. Let A_i and B_i refer to the i^{th} phases performed by clients A and B, respectively.

Lemma-1: If any of the following cases, representing all possible sequence of events, occur, then a conflict can be detected by client A (in the first two cases), or by client B (in the last two cases):

- A_2 after B_1 but before B_3
- A_2 after B_1 with B_3 in-between
- B_2 after A_1 but before A_3
- B_2 after A1 with A_3 in-between

Proof: For the first case, the temporary file uniqueness verification of A_2 will detect the temporary file placed by B_1. Thus, this conflict is detected. For the second case, there is no temporary file left from B_1 because it has already been removed by B_3. However, A_2 will still detect the conflict through the file version verification since B_3 would not have been done unless a new version is uploaded. The third and fourth cases are similar to the first two cases except that the clients are interchanged. □

Theorem-1: The proposed algorithm for the cloud-specific section is sufficient to enable at least one of any conflicting two clients to detect a conflict. *Proof:* All the cases that might occur between the phases of clients A and B are represented by the permutation of the six phases $A_1, A_2, A_3, B_1, B_2, B_3$ with the condition $Cond.1$: that phases of a client always occur in ascending order (per design of the algorithm).

The following lists all $Perm(\{A_1, A_2, A_3, B_1, B_2, B_3\}, 6)$ such that $Cond.1$ holds:

$$\begin{aligned}
&\{(A_1, A_2, B_1, A_3, B_2, B_3), (A_1, A_2, B_1, B_2, A_3, B_3),\\
&(A_1, A_2, B_1, B_2, B_3, A_3), (A_1, A_2, A_3, B_1, B_2, B_3),\\
&(A_1, B_1, B_2, B_3, A_2, A_3), (A_1, B_1, B_2, A_2, B_3, A_3),\\
&(A_1, B_1, B_2, A_2, A_3, B_3), (A_1, B_1, A_2, A_3, B_2, B_3),\\
&(A_1, B_1, A_2, B_2, A_3, B_3), (A_1, B_1, A_2, B_2, B_3, A_3),\\
&(B_1, A_1, A_2, B_2, B_3, A_3), (B_1, A_1, A_2, B_2, A_3, B_3),\\
&(B_1, A_1, A_2, A_3, B_2, B_3), (B_1, A_1, B_2, A_2, A_3, B_3),\\
&(B_1, A_1, B_2, A_2, B_3, A_3), (B_1, A_1, B_2, B_3, A_2, A_3),\\
&(B_1, B_2, B_3, A_1, A_2, A_3), (B_1, B_2, A_1, B_3, A_2, A_3),\\
&(B_1, B_2, A_1, A_2, B_3, A_3), (B_1, B_2, A_1, A_2, A_3, B_3)\} \quad (1)
\end{aligned}$$

In all of these 20 cases, at least one of the sequences mentioned in lemma-1 is true. Thus, by lemma-1, a conflict can always be detected by at least one client (A or B) or both. □

Theorem-2: If one of two conflicting clients detects a data conflict, then design objectives O_1 and O_2 are achieved.

Proof: if A detects a conflict with B on a file f, A will create and upload a unique copy f_{A_ID}. Then, we have two cases for B:

1) If B detects its conflict with A, it will upload f_{B_ID}. Thus, both f_{A_ID} and f_{B_ID} have different uploaders (O_1). Furthermore, O_2 is preserved by default as these clients will upload new files (not updating the original one).
2) If B does not detect its conflict with A (i.e., the data conflict flag is False) and hence proceeds to update the original file f, then still each file will have a unique uploader (design objective O_1). Also, O_2 is preserved for f_{A_ID} as it is new. O_2 is preserved for f since the data conflict flag is False, confirming that the parts' versions in the cloud are older than those being uploaded.

□

Note that from a distributed systems perspective, the second case, when only one client detects the conflict, does not represent consensus as the two clients do not agree on the same value (conflict existence or lack thereof). Nevertheless, O_1 and O_2 still hold.

Theorem-3: If the execution of Algorithm 2 is interrupted, design objective O_3 is still achieved.

Proof: We distinguish two cases:

1) An interruption during the execution of lines 1-10 will not affect the original parts (they will stay homogeneous).
2) An interruption during the execution of lines 11-12 leads some parts to be of a higher version than

others. In this case, a reading client can detect the different versions and continue the overwriting (renaming) process on the clouds where the renaming did not start/finish, ensuring the availability of a homogeneous set of parts.

□

Note that in the first case, the interrupted client is responsible for cleaning or completing the upload of its temporary files. This process can be initiated at startup. Until the interrupted client cleans the temporary files, other clients will perceive a conflict, and all edits to that file can still be made. However, they will be saved as another conflicted copy.

By *Theorem-1*, and *Theorem-2*, design objectives O_1 and O_2 are always met. By *Theorem-3*, design objective O_3 is always met. Hence, the proposed uploading algorithm satisfies the defined data consistency requirements in multi-cloud storage systems.

B. ON OPERATIONS ATOMICITY AND CLOUDS' CONSISTENCY MODELS

The operation of placing and removing the temporary file on a cloud storage does not need to be atomic. The verification phase of a client B can occur simultaneously with the temporary file placing phase or with the temporary file renaming phase of another client A. This is respectively shown in cases (a) and (b) in Fig.4

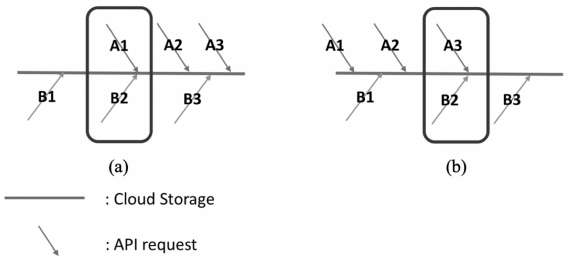


FIGURE 4. Example of concurrent phases of two clients.

The situation in the first case will be interpreted either as $(B_1, A_1, B_2, A_2, B_3, A_3)$ or $(B_1, B_2, A_1, A_2, A_3, B_3)$. While the second case will be interpreted as either $(A_1, B_1, A_2, A_3, B_2, B_3)$, or $(A_1, B_1, A_2, B_2, A_3, B_3)$. As shown earlier, in all of these four cases the data conflict will be detected and data consistency will still be maintained without the need for clouds to provide the guarantee of atomic operations.

Furthermore, the consistency model followed by each cloud provider does not affect conflict detection. For example, the *eventual consistency* model is popular among cloud storage providers. Under this model, any specific sequence of phases cannot be guaranteed to be observed by the client due to different execution (i.e., processing) time on each cloud. Despite this, the conflict is guaranteed to be detected per *Theorem-1* as the cases in (1) span all possible situations. In general, the presented method represents an application-level algorithm that is agnostic to the consistency model and operations' atomicity of the clouds.

VI. PERFORMANCE EVALUATION AND DISCUSSION

In order to evaluate the performance of the proposed method, a multi-cloud storage system is built. The system is developed using the FUSE library for Linux machines and available as open source. It provides all standard file system operations (read, write, directory listing …etc.). Also, the system includes encryption and encoding functionalities. The data (files) in the system are distributed over three public cloud storage providers (Dropbox, GoogleDrive, and Box).

The proposed method for data consistency was integrated into this multi-cloud storage system in order to test its performance. The system's overall workflow for writing a file is as follows: (1) The user issues an application-level *write* command. (2) The FUSE-defined *write* command implementation is activated, which passes the data through the encryption-partitioning-decoding pipeline illustrated in Fig. 1. The output of this pipeline is the parts and their metadata (including the version of their originating file). (3) The resulting parts, and their corresponding metadata file, are fed into the consistency algorithm (algorithm 2), whose output indicates whether the file was updated, or a new conflicted copy is created, preserving the data consistency requirements.

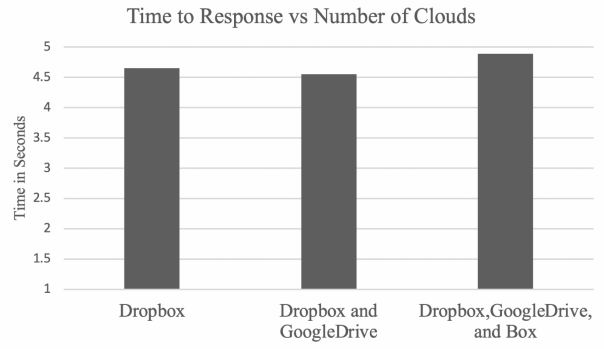
The performance evaluation was done using two machines: a main Linux machine with Intel Core i7-3770S CPU, 8 Gigabyte of RAM, 500 Gigabyte 7200 RPM hard drive, 1 Gigabit Ethernet Network running Ubuntu 16.04, and another virtual machine running with three cores of Intel core CPU 7700HQ, 4GB RAM, 64 SSD Storage, and 300 Megabits 802.11n Wireless connection. Such a setup is representative of a typical end-user device and is similar to configurations used in the literature for evaluating cloud storage systems [28].

A. CLOUDS' API REQUESTS

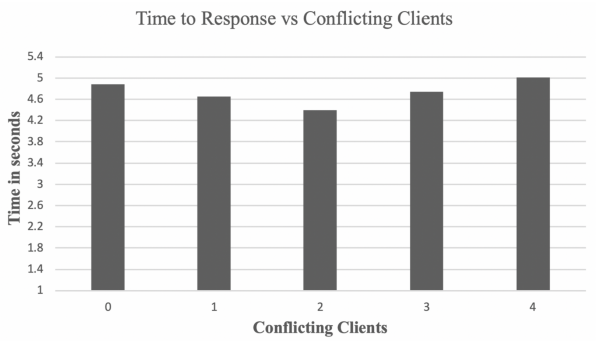
The number of Application Programming Interface (API) requests required in the proposed method is four. These calls are:

- Uploading a client-specific temporary file file
- Listing temporary file files
- Downloading metadata (to check the version of the file)
- Removing the client-specific temporary file.

The first call corresponds to the first phase (temporary file placing phase) of the cloud-specific section. The second and third calls correspond to the second phase (temporary file uniqueness and version verifications). Whereas the last call corresponds to the third phase (temporary file-renaming phase). The temporary file renaming might be done in the cloud-specific section shown in algorithm 1 (if a conflict occurs), or in the general section shown in algorithm two if there is no conflict. In all cases, this API request will be fired for each cloud. Since these calls are for the cloud-specific section, the total number of API requests needed for all used



(a) TTR vs. number of used cloud storage providers



(b) TTR vs. number of conflicting clients for the case of three clouds

FIGURE 5. TTR in different scenarios: (a) multiple clouds, (b) multiple conflicting clients.

clouds is "number of used clouds" $\times 4 = 3 \times 4 = 12$. As mentioned earlier, all cloud-specific sections run on parallel through threading.

B. TIME TO RESPONSE

In this section, we quantify the delay metric to measure the overhead added by the consistency module. We define Time to Response (TTR) metric as the time between a client's request to write/modify a file and the time of determining the status of this request based on the responses of the clouds' servers. The status of the request is the data conflict indicator. If a client receives a no conflict indicator, the client will start uploading the new file's parts to their respective clouds, overwriting the older parts. On the other hand, if a client detects a conflict, it will create a new file (conflicted copy) and upload its parts to the respective clouds, without affecting the parts of the original file. Thus, the client will be able to start writing (uploading) file parts to the clouds after the end of the second phase of the cloud-specific sections, or in other words (after TTR).

As shown in Fig. 5a, the average TTR is approximately 4.7 seconds. Since the cloud-specific sections of the algorithm can run in parallel, the TTR is roughly the same when tested on one, two, and three public cloud providers. Thus, the proposed algorithm can be scaled to multi-cloud storage systems that utilize many cloud providers due to the thread-friendly design of the cloud-specific section.

Additionally, TTR is roughly the same, whether there is a data conflict between any number of clients or there is not any, which means that the proposed method can also be scaled with respect to the number of users. Fig. 5b shows the TTR when the main testing machine is not conflicting with any other machine, and when it is conflicting with one, two, three, and four other secondary machines. The similar delay is because the existence of a conflict does not change the complexity of the proposed method; the effect of a conflict existence is merely writing a different copy. Hence, provided that cloud providers API's can process the requests from additional clients, a higher number of conflicting clients should not affect the method's latency.

C. COMPARISON WITH BASE CASES

In this section, we present end-to-end delay (i.e., including TTR and data upload time) comparison between the case when the data consistency module is not activated versus the case when it is activated. Fig. 6 shows actual times for uploading files in the designed multi-cloud storage system with and without the proposed data consistency method. These times include the partitioning, encoding, and encryption of the data parts. As shown in the figure, the overhead added by the data consistency algorithm is roughly 5 seconds, including the TTR and the time required for temporary file renaming (third phase). The additional 1 second that exists in the case of conflict is mainly due to the conflict handling mechanism (copying data to the new conflicted copy) of our system, and not due to the algorithm itself since, as shown earlier, the TTR is the same whether there is a conflict or not.

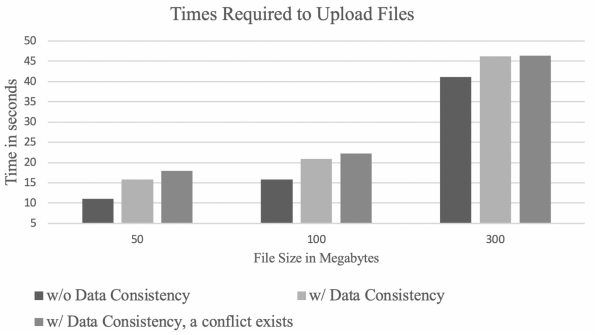


FIGURE 6. Overhead in uploading data to multi-cloud storage system.

D. COMPARISON WITH OTHER TECHNIQUES

Table 2 compares the performance in terms of the upload latency with the two previous techniques reported in MetaSync [25] and SaveMe [28]. These two techniques are selected as representatives of similar previous work since they are the most recent multi-cloud data consistency solutions available in the literature that support passive servers, use non-communicating clients, and execute all operations on the client-side (i.e., no third-party service to handle consistency).

TABLE 2. Latency overhead comparison (reported averages in seconds).

Reference	**Upload overhead** (seconds)
MetaSync [25]	≈ 5.1
SaveMe [28]	$\approx 1 - 20$ (Depending on the deadlock deadline)
This work	≈ 4.7

Although the proposed method has better performance on average, the results illustrate an approximately similar performance. The main differentiator between these approaches is the necessary assumptions, which are minimized for our proposed method, making it more practical. These assumptions are discussed in the related work section and are summarized in Table 1.

E. CLOUD STORAGE SERVICE OUTAGE AND DATA CONSISTENCY

The proposed algorithm detects data conflicts even if the available clouds are not the same for different users, provided that they overlap in at least one cloud. If $n > t$, this is always guaranteed. For example, for a client A, if only the two clouds $C1$ and $C2$ are available from the used three $C1$, $C2$, and $C3$, and for another client B, the available clouds are $C2$ and $C3$. The data consistency is still guaranteed since they overlap in $C2$ where the cloud-specific section related to $C2$ would still be able to perform all the verifications.

VII. CONCLUSION

In this paper, we discussed multi-cloud storage systems and their significant advantages in establishing trust in the cloud. The paper focuses on addressing the concurrent data access issue and reasons why it is especially challenging in such systems compared to conventional storage systems. Various previous solutions to this issue were discussed. The paper offered a useful formal definition of data consistency and data conflicts in a multi-cloud storage system and proposed a novel method to detect data conflicts and maintain data consistency. The method:

- Does not require inter-client or inter-server communication,
- Avoids the reliability and security issues associated with a single point of failure as it does not require the help of any server (a server-less system that runs fully on the client's machine),
- Utilizes passive cloud storage services,
- Is scalable with respect to the number of clouds,
- Is scalable with respect to the number of users, and
- Works as long as users of a multi-cloud storage system overlap in using at least one cloud.

Experimental results on real cloud systems show an API calls-delay of approximately 4.5 seconds before uploading data to the multiple clouds. The proposed algorithm is best suited for a multi-cloud storage system that requires data consistency while also being scalable and tolerant to different cloud failures. Future work might consider utilizing local caching or API call optimization to minimize the delay.

REFERENCES

[1] J. Domingo-Ferrer, O. Farràs, J. Ribes-González, and D. Sánchez, "Privacy-preserving cloud computing on sensitive data: A survey of methods, products and challenges," *Comput. Commun.*, vols. 140–141, pp. 38–60, May 2019.

[2] R. Nachiappan, B. Javadi, R. N. Calheiros, and K. M. Matawie, "Cloud storage reliability for big data applications: A state of the art survey," *J. Netw. Comput. Appl.*, vol. 97, pp. 35–47, Nov. 2017.

[3] Y.-T. Lee, W.-H. Hsiao, Y.-S. Lin, and S.-C.-T. Chou, "Privacy-preserving data analytics in cloud-based smart home with community hierarchy," *IEEE Trans. Consum. Electron.*, vol. 63, no. 2, pp. 200–207, May 2017.

[4] E. Torres, G. Callou, G. Alves, J. Accioly, and H. Gustavo, "Storage services in private clouds: Analysis, performance and availability modeling," in *Proc. IEEE Int. Conf. Syst., Man, Cybern. (SMC)*, Oct. 2016, pp. 3288–3293.

[5] P. Li, S. Guo, T. Miyazaki, M. Xie, J. Hu, and W. Zhuang, "Privacy-preserving access to big data in the cloud," *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 34–42, Sep. 2016.

[6] J. Li and B. Li, "Erasure coding for cloud storage systems: A survey," *Tsinghua Sci. Technol.*, vol. 18, no. 3, pp. 259–272, Jun. 2013.

[7] *That Cloud Holding Your Personal Photos? Its Security is Really a Low-Qualified Locker | IEEE Computer Society*. Accessed: Aug. 14, 2020. [Online]. Available: https://www.computer.org/publications/tech-news/research/photo-security-privacy-cloud-computing

[8] K. M. Khan and Q. Malluhi, "Trust in cloud services: Providing more controls to clients," *Computer*, vol. 46, no. 7, pp. 94–96, Jul. 2013.

[9] D. Seo, S. Kim, and G. Song, "Mutual exclusion strategy in a cloud-of-clouds," in *Proc. IEEE Int. Conf. Consum. Electron. (ICCE)*, Jan. 2017, pp. 124–125.

[10] S. Ghosh, *Distributed Systems: An Algorithmic Approach*. Boca Raton, FL, USA: CRC Press, 2014.

[11] R. A. Campêlo, M. A. Casanova, D. O. Guedes, and A. H. F. Laender, "A brief survey on replica consistency in cloud environments," *J. Internet Services Appl.*, vol. 11, no. 1, p. 1, Dec. 2020.

[12] D. Bermbach and J. Kuhlenkamp, "Consistency in distributed storage systems: An overview of models, metrics and measurement approaches," in *Networked Systems* (Lecture Notes in Computer Science), vol. 7853. Berlin, Germany: Springer, 2013, pp. 175–189.

[13] M. Schnjakin and C. Meinel, "Evaluation of cloud-RAID: A secure and reliable storage above the clouds," in *Proc. 22nd Int. Conf. Comput. Commun. Netw. (ICCCN)*, Nassau, Bahamas, Jul. 2013, pp. 1–9. [Online]. Available: http://ieeexplore.ieee.org/document/6614137/

[14] H. C. H. Chen, Y. Hu, P. P. C. Lee, and Y. Tang, "NCCloud: A network-coding-based storage system in a cloud-of-clouds," *IEEE Trans. Comput.*, vol. 63, no. 1, pp. 31–44, Jan. 2014.

[15] G. Zhao, M. G. Jaatun, A. Vasilakos, A. A. Nyre, S. Alapnesy, Q. Yue, and Y. Tang, "Deliverance from trust through a redundant array of independent net-storages in cloud computing," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, Apr. 2011, pp. 625–630.

[16] M. Jaatun, G. Zhao, A. V. Vasilakos, Å. A. Nyre, S. Alapnes, and Y. Tang, "The design of a redundant array of independent net-storages for improved confidentiality in cloud computing," *J. Cloud Comput., Adv., Syst. Appl.*, vol. 1, no. 1, p. 13, 2012.

[17] D. Decasper, A. Samuels, and J. Stone, "Redundant array of independent clouds," U.S. Patent 2012 0 047 339 A1, Feb. 23, 2012. [Online]. Available: https://patents.google.com/patent/US20120047339/en

[18] D. Li and Y. Zhou, "A secure and reliable hybrid model for cloud-of-clouds storage systems," in *Proc. IEEE 22nd Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2016, pp. 1157–1162.

[19] A. Sampaio and N. Mendonça, "Uni4Cloud: An approach based on open standards for deployment and management of multi-cloud applications," in *Proc. 2nd Int. Workshop Softw. Eng. Cloud Comput. (SECLOUD)*, Honolulu, HI, USA, 2011, p. 15. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1985500.1985504

[20] P. Shen, W. Liu, Z. Wu, M. Xiao, and Q. Xu, "SpyStorage: A highly reliable multi-cloud storage with secure and anonymous data sharing," in *Proc. Int. Conf. Netw., Archit., Storage (NAS)*, Aug. 2017, pp. 1–6.

[21] R. Pottier and J.-M. Menaud, "TrustyDrive, a multi-cloud storage service that protects your privacy," in *Proc. IEEE 9th Int. Conf. Cloud Comput. (CLOUD)*, Jun. 2016, pp. 937–940.

[22] P. Viotti, D. Dobre, and M. Vukolić, "Hybris: Robust hybrid cloud storage," *ACM Trans. Storage*, vol. 13, no. 3, Oct. 2017, Art. no. 27, doi: 10.1145/3119896.
[23] J. Lejeune, L. Arantes, J. Sopena, and P. Sens, "Service level agreement for distributed mutual exclusion in cloud computing," in *Proc. 12th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (ccgrid)*, May 2012, pp. 180–187.
[24] S. Han, H. Shen, T. Kim, A. Krishnamurthy, T. Anderson, and D. Wetherall, "MetaSync: File synchronization across multiple untrusted storage services," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 83–95.
[25] S. Han, H. Shen, T. Kim, A. Krishnamurthy, T. Anderson, and D. Wetherall, "MetaSync: Coordinating storage across multiple file synchronization services," *IEEE Internet Comput.*, vol. 20, no. 3, pp. 36–44, May 2016.
[26] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood, "Cloud types for eventual consistency," in *Proc. Eur. Conf. Object-Oriented Program.* Berlin, Germany: Springer, 2012, pp. 283–307.
[27] G. Song, S. Kim, and D. Seo, "SaveMe: Client-side aggregation of cloud storage," *IEEE Trans. Consum. Electron.*, vol. 61, no. 3, pp. 302–310, Aug. 2015.
[28] D. Seo, S. Kim, and G. Song, "Mutual exclusion method in client-side aggregation of cloud storage," *IEEE Trans. Consum. Electron.*, vol. 63, no. 2, pp. 185–190, May 2017.

NARAM MHAISEN received the B.Sc. degree (Hons.) in computer engineering from Qatar University, where he is currently pursuing the M.Sc. degree in computing. He is also a Graduate Research Assistant with the KINDI Computing Research Centre. His current research interests include distributed and multiagent systems.

QUTAIBAH M. MALLUHI (Member, IEEE) received the B.S. and M.S. degrees in computer engineering from KFUPM, Saudi Arabia, and the M.S. and Ph.D. degrees in computer science from the University of Louisiana, Lafayette. He was the Head of the Department from 2006 to 2012. He was the Director of the KINDI Center for Computing Research, Qatar University (QU), from 2012 to 2016. He was a Professor with Jackson State University and a Research Faculty Member with the Lawrence Berkeley National Laboratory. He was also a Co-Founder and a CTO with Data Reliability Inc. He is currently a Professor with the Department of Computer Science and Engineering, QU. He received several honors and awards, including the QU Research Award, the JSU Technology Transfer Award, the Mississippi MURA, and the JSU Faculty Excellence Award.