electronics

MDPI

Article

Cloud Storage Service Architecture Providing the Eventually Consistent Totally Ordered Commit History of Distributed Key-Value Stores for Data Consistency Verification

Beom-Heyn Kim and Young Yoon *

Department of Computer Engineering, Hongik University, 94 Wowsan-ro, Sangsu-dong, Mapo-gu, Seoul 04068, Korea; beomheyn.kim@gmail.com

* Correspondence: young.yoon@hongik.ac.kr; Tel.: +82-02-320-1659

Abstract: Cloud storage services are one of the most popular cloud computing service types these days. Various cloud storage services such as Amazon S3, DropBox, Google Drive, and Microsoft OneDrive currently support billions of users. Nevertheless, data consistency of the underlying distributed key-value store of cloud storage services remains a serious concern, making potential customers of cloud services hesitate to migrate their data to the cloud. Researchers have explored how to allow clients to verify the behavior of untrusted cloud storage services with respect to consistency models. However, previous proposals are limited because they rely on a strongly consistent history server to provide a totally ordered history for clients. This work presents Relief, a novel cloud storage service exposing an eventually consistent totally ordered commit history of the underlying distributed key-value store to enable client-side data consistency verification for various consistency models. By empirically evaluating our system, we demonstrate that Relief is an efficient solution to overcome the limitation of previous approaches.

Keywords: cloud computing; cloud storage services; cloud database systems; distributed key-value store; data consistency; verification

Citation: Kim, B.-H.; Yoon, Y. Cloud Storage Service Architecture Providing the Eventually Consistent Totally Ordered Commit History of Distributed Key-Value Stores for Data Consistency Verification. *Electronics* **2021**, *10*, 2702. https://doi.org/10.3390/electronics10212702

Academic Editor: George Angelos Papadopoulos

Received: 27 September 2021
Accepted: 1 November 2021
Published: 5 November 2021

1. Introduction

One of the most popular forms of cloud computing services today is cloud storage services (CSS). Various CSSs include Amazon S3, DropBox, Google Drive, Microsoft OneDrive, and Amazon DynamoDB. The recent survey projects the cloud storage service market to grow from 50.1 billion US Dollars in 2020 to 137.3 billion US Dollars by 2025, at a compound annual growth rate of 22.3% [1]. Because cloud storage services distribute and replicate data at a global scale, users with internet access can work on the same dataset to collaborate with others anywhere and anytime. Moreover, users do not need to worry about complications involved in managing physical storage servers since the cloud service provider's responsibility is to ensure that the underlying infrastructure works as expected to serve their customers faithfully.

Data consistency is the fundamental and essential property of the underlying *distributed key-value stores* (DKVS) of CSSs, which client applications depend on [2]. It defines the rules for clients to determine whether their CSS correctly behaves or not with respect to read or write operations. Modern DKVSs implement *consistency models* defining the exact nature and the meaning of data consistency in terms of *consistency guarantees* for clients. The violation of those guarantees puts clients at risk of making decisions based on inconsistent data, leading to a severe consequence such as the motivational example given in Section 2.

Therefore, researchers have explored various techniques for clients to verify data consistency provided by a storage service (hereafter, "consistency verification"). Clients may record their operations as a commit history and perform complex graph-based analysis

based on the time when a request is issued and the time when the corresponding response is received [3,4]. Yet, the commit history constructed by clients without any information from servers is not accurate. Thus, such methods are slow, especially when there are many concurrent writes because of uncertainty in figuring out the actual commit order of operations. Other researchers have utilized such a *commit history* of operations exposed by the server. They represent the history as hash chains where each operation is bound with previous ones via cryptographic hash [5–12]. However, with those proposals, the cost is incurred by computing cryptographic hashes and resolving conflicts to merge forked chains. To solve this issue, some other works have used a history more generally represented as a log recording operations committed to support checks for multiple consistency models on a single platform [13,14]. Nevertheless, they are limited because their history server (HS) is strongly consistent to assign a monotonically increasing global sequence number to serialize each operation. Therefore, the HS component remains a potential performance bottleneck.

In this work, we develop a novel *cloud storage service architecture providing a log containing eventually consistent totally ordered commit history for the client-side data consistency verification.* After closely analyzing the cloud storage service architecture of previous proposals, we found there are two challenges involved in constructing a log containing a totally ordered history for consistency verification. First, the order of entries in the log should be totally ordered, and therefore each operation must be uniquely identifiable. If there are concurrent operations not explicitly ordered, clients may not be able to see some operations in a log that are overwritten by a concurrent operation. Second, the total order of log entries should be the reflection of the actual commit order of operations determined by the underlying DKVSs. Although the operations recorded in the log may not be ordered as committed. Therefore, it is up to the clients to figure out the actual order using the information included in the log.

To overcome the limitation of previous cloud storage service architectures, we devised a novel mechanism, *conflict-free replicated history update* (CRHU), to construct a log containing totally ordered history in an eventually consistent manner. The critical observation is that *utilizing versions that DKVSs internally use to order all operations* enables *the eventual construction of a log containing totally ordered history reflecting the actual commit order of operations*. Exploiting this observation, our technique distributes and replicates log replicas. Log replicas are concurrently updated with a key-value pair where the key is a version and the value is the meta-data of an operation. Updates made to those replicas are eventually replicated. For concurrent operations, we merge them into a totally ordered sequence with the order determined based on versions. Because versions used as keys cause no conflict, no overwriting occurs among concurrent updates, and all replicas eventually converge to reflect the same totally ordered history. In addition, as versions determine the actual commit order of operations assigned by DKVSs, the order of log entries reflects the actual commit order of corresponding operations. Then, the resulting log can be used by clients to conduct data consistency verification for an arbitrary consistency model.

To demonstrate the effectiveness of the proposed approach, we implemented CRHU in our prototype CSS named Relief. The core component of Relief is a distributed system sitting between clients and a DKVS to intercept requests from clients and mediate their access to the underlying DKVS. After the DKVS commits those requested operations, Relief receives versions of the committed operations along with DKVS's responses. Using those versions, Relief updates log replicas to honour the actual commit order. Our prototype uses versions that are timestamps obtained from synchronized physical clocks. We evaluated our prototype system and found we could gain 100% throughput and 200% read latency improvements compared to a strongly consistent approach. Also, we show the speed of consistency verification using logs of Relief can scale linearly as reading percentages and operations increase.

Contributions

The main contribution made by this work is its novel cloud storage service architecture that can construct a commit history log containing entries ordered reflecting the actual commit order of the underlying distributed key-value store systems while not strictly serializing concurrent updates of the log but allowing eventual determination of their total order. This has not been presented by any of previously known cloud storage service architectures enabling client-side data consistency verifications. The major challenge blocking previous works to overcome their limitation is that building the commit history log in an eventual manner without dragging the performance of the underlying distributed key-value stores to reflect the actual commit order into the log. The key enabler we found out is that the underlying distributed key-value stores can expose their internal mechanisms used to determine the actual commit order of operations for history servers to employ to construct the eventually consistent commit history log reflecting the actual commit order. This observation frees the cloud storage service architecture from the undesirable requirement degrading the performance of service as a whole.

Alongside the main academic contribution, we made following specific contributions:

- We analyze the fundamental limitation of previous proposals enabling client-side data consistency verification.
- This is the first work to explore how to provide an eventually consistent totally ordered commit history for client-side data consistency verification.
- We built a prototype system, Relief (https://github.com/Kaelus/Relief, accessed on 27 September 2021), and empirically evaluated its performance compared to previous approaches.
- We integrate a consistency oracle with our prototype and demonstrate the efficiency of the white-box consistency verification using the history provided by Relief.

The rest of our paper is structured as follows. We present the background knowledge in Section 2. Then, we describe the problem we solve in Section 3. Relief's architecture is discussed in Section 4, and we report the evaluation results in Section 5. In Section 6.1, we compare our work against related works. Finally, we conclude in Section 7.

2. Background

In this section, we provide prerequisite knowledge needed to understand the rest of this paper. We explain why it is vital to do consistency verification for consistency models of CSSs. Then, we summarize the advance made in the field of consistency verification. In addition, we discuss distributed events and the methods to track causality and figure out their order.

2.1. Motivation for Consistency Verification

One example where consistency violations can lead clients to make a critical mistake is an authorization service relying on an access control list stored on a strongly consistent CSS. A client may share a file with another party, but the client decides to add information that should be confidential to others. The client sends a request to the CSS to revoke the read permission of the unwanted party for their shared file. However, the revocation operation may be delayed so that it might not be replicated to every replicated node of the CSS on time. For instance, such a situation may occur if there was an unexpected delay in the network caused by a software or hardware failure (not to mention an abrupt and unexpected denial-of-service attack with malicious intent) that interrupts the replication of operations between replicated nodes of the CSS. Meanwhile, the client adds the confidential information to the shared file, and the unwanted party tries to read it at the same time. The access control list becomes inconsistent because the revocation operation has not been replicated to every server yet. If the authorization service makes a wrong decision based on the stale access control list (the one from the server that has not replicated the revocation operation yet), then the unwanted party may successfully read the shared file, which breaches the data confidentiality.

Therefore, clients need the ability to verify their CSSs for consistency models, assuming those CSSs are untrusted from the client perspective. That is, they need the means to perform client-side consistency verification. Clients only know about their own operations but do not know about others'. Such a history perceived by each client is called *view*. Clients may not be able to obtain the history of all operations without the support from CSSs. Then, the consistency verification requires a complex graph-based analysis based on when a request is issued and when the corresponding response is received. Such verification method can be slow when there are many concurrent operations since there is uncertainty involved in figuring out the actual order of operations committed by CSSs [3,4]. A more efficient algorithm is enabled with the support from CSSs exposing the history of operations on the request from clients, reflecting the actual order of operations they committed [13]. Because the order is explicitly stated in the history declared by CSSs, consistency verification with the declared history is more deterministic, therefore more straightforward, accurate and efficient.

2.2. Consistency Verification

Consistency verification is a technique checking if all operations are validly ordered and become visible under the given consistency model. There are two aspects in it regarding what makes the correct and incorrect operations concerning consistency models. For read operations, clients must see the writes that are valid under the given consistency model. For example, suppose the given consistency model is linearizability [15], and there are two writes performed before issuing a read operation. Those two operations are namely w_1 and w_2, where the start time of w_2 is later than the end time of w_1. According to the definition of linearizability, any subsequent read operation must see the value written by w_2, as the value written by w_1 must be overwritten by the value written by w_2. If a subsequent read operation sees the value written by w_1, then the read operation violates linearizability.

Consistency verification is a challenging task when there are concurrent operations and a storage server does not tell clients about the actual commit order of operations. The actual commit of concurrent operations may occur anytime between the request's start and end times due to variation in networking and processing delays involved at any point on the network path between clients and the storage medium. Therefore, clients only know about the time when they issue requests and the time when they receive responses. Without knowing the actual commit order, consistency verification must involve exploring all possible permutations of operations based on the request's start and end times. Jepsen is the state-of-the-art testing tool looking for consistency bugs in distributed databases [4]. However, its default consistency verification does not use the actual commit order. Therefore, their verification is black-box approach which explores every possible interleaving of concurrent operations until it finds a precise ordering of operations that is valid under the given consistency model. Trivially, its computational complexity grows exponentially as the number of concurrent operations increases.

Authors of SUNDR [5] showed that the *fork consistency* is the strongest guarantee we can achieve when the storage server is untrusted and lying about the actual ordering of operations. Unless clients exchange their views constantly, it is impossible to achieve even the weakest consistency model, *eventual consistency*. They briefly hint at how such an incorrect state can be detected. One method is the existence of an online trusted client. If such a trusted client, so-called "timestamp box," exists, it can periodically request the untrusted server provide the operational history. Then, the timestamp box attests the provided history by signing the history. Finally, the timestamp box can send the signed history on the server so all other clients can see it. Assuming cryptography used for the digital signature is strong enough, the untrusted server cannot forge the history once the history is provided to the timestamp box. Another method is having clients broadcast the partial history they know about. If a conflict in the partial history is detected using version vectors, a forked history is found. Depot [6] explores the way to use peer-to-peer message

broadcasts to detect such a fork and additionally proposes how to merge the forked history by treating forks as conflicts of concurrent operations.

Caelus [13] is designed for battery-powered devices interacting through CSSs. Target devices of Caelus cannot directly communicate each other as they are behind the firewalls blocking any incoming connection. Also, those devices can become online and offline, spontaneously. Thus, none of those devices are suitable to be a timestamp box. Instead, Caelus extends the timestamp box and devises an *attestor*. Unlike the timestamp box, the attestor can dynamically migrate from one device to another. While SUNDR has pioneered and proposed a way to detect a fork, Caelus has advanced the fork detection for the newly emerging class of computing devices that are the battery-powered devices. Moreover, all previous works primarily rely on collision-resistant hashes cryptographically binding different versions of the snapshot of the dataset, which makes their check computationally expensive and inflexible as focusing on verifying a specific consistency model only, such as fetch-modify consistency in SUNDR or fork-join-causal-consistency in Depot. Meanwhile, Caelus employs the log containing the totally ordered history of operations and, therefore, supports more lightweight and dynamic consistency verification for multiple consistency models.

2.3. Clocks and Ordering of Distributed Events

DKVSs use clocks to obtain the timestamp for each committed operation to establish a causal relation with other operations that happened before across distributed replicas. For example, Lamport clock [16] is the very first logical clock that can be used to find out the causal order of events. It gives a binary relation for a pair of events. If an event A happens before the other one B, then the timestamp of A is smaller than that of B. There may be some pair of operations whose order cannot be established as the Lamport clock reveals only the partial causal order.

However, using timestamps from the Lamport clock, one can still create a total order by determining the winner of conflicting concurrent operations through the straightforward combination of the timestamp value and another readily available extra information such as a node ID. Although such a total order does not preserve the actual real-time order, we can still figure out the order of operations assigned by the underlying DKVS.

Similarly, vector clock [17–19] can be used to track causality more comprehensively. Vector clocks are used by a real-world key-value store Dynamo [20].

Physical clocks have been used as well to determine the winner in other systems like Cassandra, DynamoDB and Spanner based on the last-write-win conflict resolution [21–23]. Moreover, BigTable allows clients to provide unique timestamps to determine orders [24], which is similar to letting clients assign orders in Caelus, but used for the internal conflict-resolution under weak consistency.

Although mechanisms exist to create a total order of operations, they have never been used to provision a totally ordered history for consistency verification.

3. Problem Statement

In this section, we explain the problem Relief solves in detail. We more clearly specify our target environment. Subsequently, we elaborate on the limitations of existing approaches. First, previous solutions rely on strong consistency when updating the history, making the HS a potential performance and scalability bottleneck. Second, the total order in the history should reflect the order of operations committed by a DKVS but achieving it is not trivial due to variable delays for networking and processing.

3.1. Target Environment

Figure 1 illustrates our target environment. A target CSS architecture has 2 components: *cloud server* (CS) and *distributed key-value store* (DKVS). Clients are running on user's endpoints and perform read and write operations via `GET` and `PUT` interface. Servers running on a CSS mediate access to the DKVS to read and write data. Also, servers update

a history regarding committed operations. Among clients, an attestor may read the history via `READ_HISTORY`, attest it and write the attestation back to the DKVS via `WRITE_ATTEST`. Other clients may download attestations and the corresponding sections of the history via `READ_ATTEST` and perform consistency verification. For more detailed specification of the interface, refer to Section 4.1.

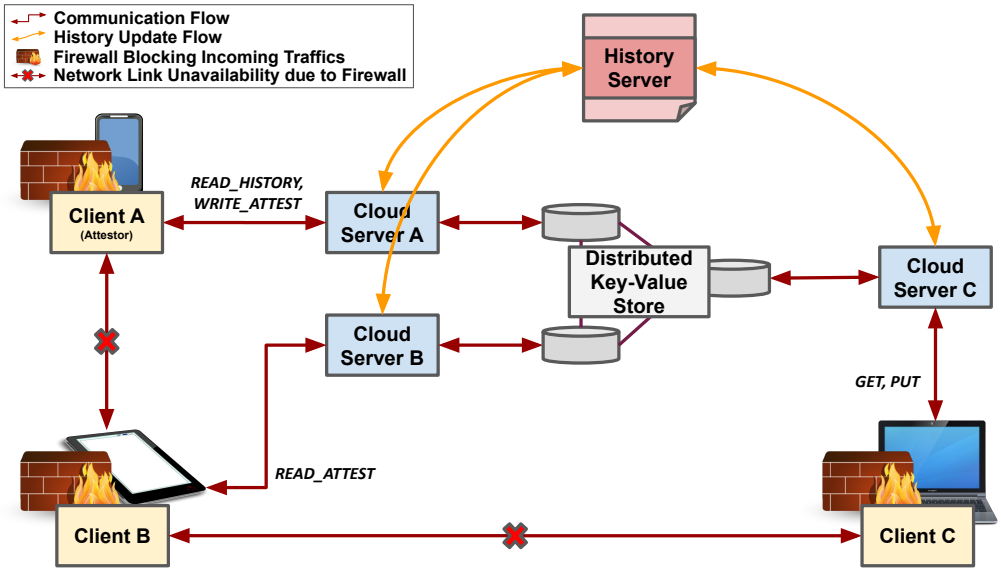

Figure 1. Target Environment of Relief.

We assume clients are endpoints behind firewalls where the default setting denies all inbound traffic for security reasons. For example, Windows Firewall does so [25]. Clients are running on heterogeneous endpoint devices running on batteries. Therefore, they may be on and off spontaneously due to intermittent disconnection from the network while traveling. This setting makes previous approaches to support client-side data consistency verification challenging, because clients cannot directly send and receive their views to verify consistency of operations by analyzing the history of operations. The solutions to this issue have been adding a history server component on the cloud storage service architecture that collects and distributes clients' views on behalf of clients.

On the other hand, we suppose the CSS is highly available, unlike clients that can churn. Clients have some storage space to remember the history of their reads and writes. Also, clients have synchronized clocks among them. One example can be a well-known NTP clock synchronization protocol. Although NTP protocol is not as accurate as the mechanism used by Spanner [23], we consider it precise enough to support tens of target endpoint devices. For the large-scale deployment, one may replace clocks synchronized with NTP protocol with more tightly synchronized clocks.

In Relief's target environment, clients are the trusted components while any other components, including cloud services, are untrusted. Cloud services are composed of many moving parts, each containing software or hardware bugs that may manifest at the worst timing. In addition, there may be malicious attackers, both external and internal. Both bugs and attacks may affect the CSS to violate consistency models. With this, any application's assumption can break, and critical mistakes can occur. The scenario mentioned above involving the faulty authorization service described in Section 2.1 is merely one example.

3.2. A History Server Using Strong Consistency

To support consistency verification for clients, each entry in the log containing the totally ordered history must be uniquely identifiable. To provide a service to many clients with some degree of fault-tolerance, it is natural to build the HS as a distributed replicated system. With this, as multiple clients access data simultaneously, concurrent updates

made to the history are inevitable. Therefore, there can be several entries appended to the history by those concurrent updates. Since clients will request the specific range of entries in the history, entries in the history must be indexed with identifiers that clients can use to specify the exact section of the history. If such identifiers are not unique, several entries concurrently updated will be assigned with a common identifier. Then, when clients request the section of the history, some entries will be omitted because updates are overwritten by a concurrent update assigning the common identifier. The resulting history segment will be incomplete and lead to false detection of consistency violation.

Existing approaches employ a monotonically increasing global sequence number. Strong consistency models, such as strict serializability, linearizability, and sequential consistency, can help implement it. The implementation of strong consistency models needs to use synchronous coordination among replicas to serialize operations upfront. Serializing operations upfront guarantees to create the total order of operations instantly. Hence, leveraging this instant total ordering guarantee allows each entry to be uniquely assigned with a monotonically increasing global sequence number upon each update. For instance, Caelus requires at least sequential consistency, which ensures that there exists a total order of operations for each client.

Although using strong consistency allows identifying each entry in the history uniquely, it also makes the HS become a potential bottleneck for performance and scalability, as illustrated in Figure 2. Caelus assumed the HS to be replicated and run on some coordination protocol, such as consensus protocols or primary-backup replication protocols, so that it can be fault-tolerant and strongly consistent enough to provide a totally ordered history log. As more operations are ingested into the service, the HS architecture's synchronous and centralized nature can be a source of performance and scalability bottleneck.

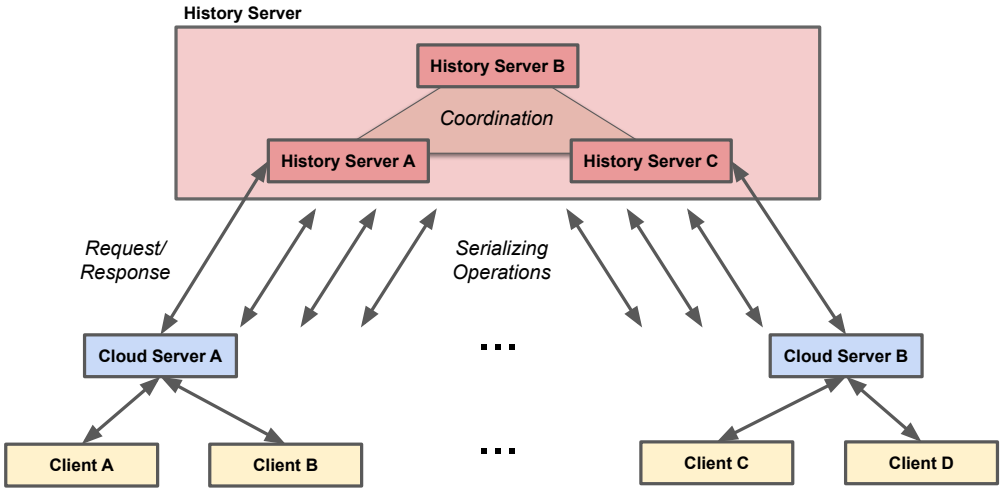

Figure 2. A History Server using Strong Consistency.

3.3. Reflecting a Commit Order in the History

To perform consistency verification by analyzing the history, it is necessary that the history should reflect the order of operations committed by a DKVS. Because the DKVS and its HS run asynchronously, it is not trivial to reflect the order of DKVS to the order in HS. This problematic situation can occur due to various networking and processing delays at any point between the clients sending requests and the components of a CSS. Figure 3 gives an illustrative example. In the example, if the true order committed by the DKVS is as follow: `PUT(x,1)`, `PUT(x,2)` and `GET(x)2`, it is valid under linearizability. However, if the order captured in the history was: `PUT(x,2)`, `PUT(x,1)` and `Get(x)2` because updating the history for `PUT(x,1)` got delayed due to a slumping network, then consistency violation is falsely accused.

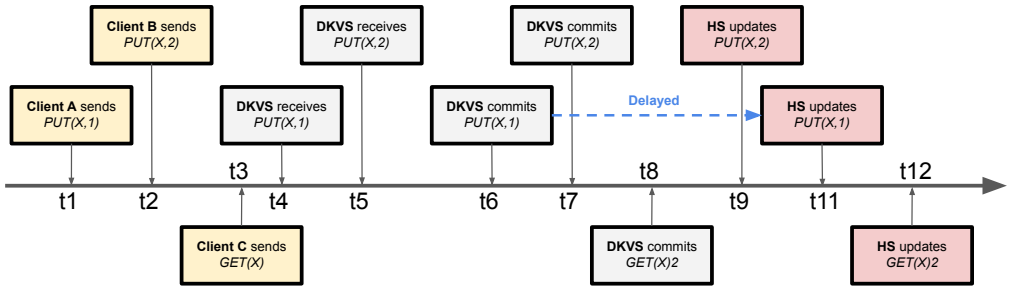

Figure 3. History Updates Out-of-Order Example.

The previous proposal, Caelus, solves this problem by having clients attach a timestamp whenever it sends a request. Then, the history entry contains the request metadate including the timestamp. Then, when a client receives a section of the history via `READ_HISTORY`, it sorts the entries based on the timestamp as depicted in Figure 4. A client will request the section of the history specified with the start global sequence number g1 and the end global sequence number g3. Then, the client receives the raw history segment. Based on timestamps on each entry, the client sorts entries and then performs consistency verification. Now, because the order in the history is sorted as: $PUT(x,1)$, $PUT(x,2)$ and $GET(x)2$, there is no false consistency violation detected under linearizability.

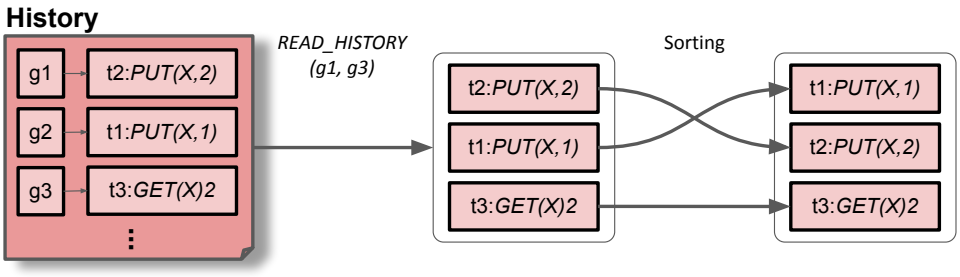

Figure 4. Client sorts the entries based on timestamps attached on each operation request.

However, note that the reordering due to a variable processing and networking delay can occur not only between a DKVS and the history server, but also between clients and the DKVS, as shown in Figure 5. In this example, although clients send their requests in the order: $PUT(x,1)$, $PUT(x,2)$ and $GET(x)$, a DKVS commits in a different order: $PUT(x,2)$, $PUT(x,1)$ and $GET(x)1$ due to the unexpectedly long delay for receiving $PUT(x,1)$. When a client sorts based on the timestamps obtained when clients send their requests, it got the sorted history ordered as: $PUT(x,1)$, $PUT(x,2)$ and $GET(x)1$. Thus, it will raise an alarm for consistency violation of linearizability.

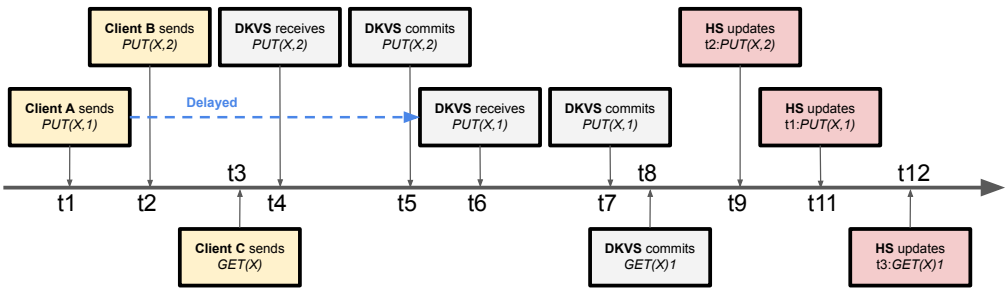

Figure 5. Commit Out-of-Order Example.

One possible solution is enforcing the DKVS commit operations requested by clients based on their timestamps attached on requests as illustrated in Figure 6. The DKVS

waits until all previous operations are committed before committing each operation. Then, both the orders reflected in the history and the DKVS's commit order are dictated by timestamps attached to the clients' requests. However, this solution leads to a significant unnecessary delay before committing operations. An alternative method is taking an opportunistic approach where the DKVS commits operations as their requests are arrived but aborts any operations that are issued earlier but arrived later than previously committed operations. Nevertheless, it is trivial to see that this approach will unnecessarily abort many operations by the DKVS. Relief's approach is using versions that the underlying DKVSs use to determine the actual commit order, so it does not suffer from those issues.

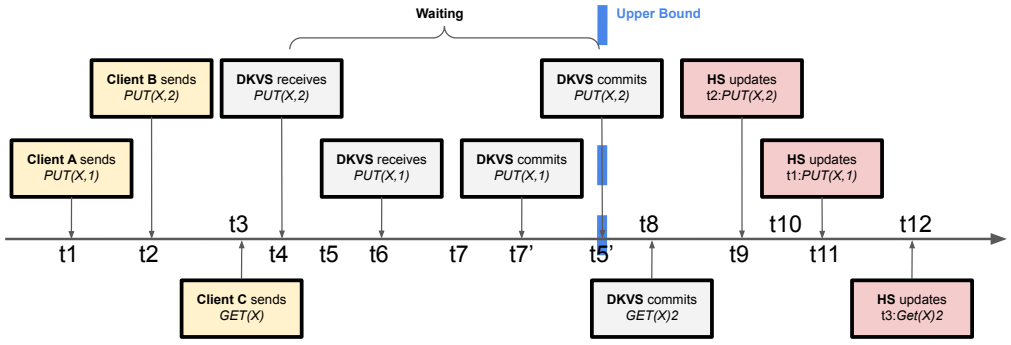

Figure 6. Delayed Commit to wait for the Upper Bound of Network Delivery Time. By waiting long enough, a DKVS can commit operations in the order that is dictated by clients' timestamps. With this, any reordering is the indication of malicious delay.

Formal Description of the Problem. Suppose we have a client, C_i, issuing requests: $R_0 = \{r_{00}, r_{01}, \ldots, r_{0n}\}$. Each issued request, r_{0i}, flows goes through C_i, a cloud server, S_i and finally a history log, H_i:

$$C_i \xrightarrow{r_0 i} S_i \xrightarrow{r_0 i} H_i$$

If there is another client, C_j, issuing requests: $R_1 = \{r_{10}, r_{11}, \ldots, r_{1n}\}$, and its request, r_{0j}, flows through the C_j, another cloud server, S_j, and finally a history log, H_j:

$$C_j \xrightarrow{r_0 j} S_j \xrightarrow{r_0 j} H_j$$

r_0x issued by C_i at time t_{x0} and r_1y issued by C_j at time t_{y0}, r_0x arrives at S_i at time t_{x1} and r_1y arrives at S_j at time t_{y1} and r_0x arrives at H_i at time t_{x2} and r_1y arrives at H_j at time t_{y2}. However, as depicted by Figure 5:

$$t_{x0} < t_{y0} \nRightarrow t_{x1} < t_{y1}$$

$$t_{x1} < t_{y1} \nRightarrow t_{x2} < t_{y2}$$

and

$$t_{x0} < t_{y0} \nRightarrow t_{x2} < t_{y2}$$

Therefore, there is no guarantee to infer the order of operations between networked components of the CSS. The solution described earlier by Figure 6 is adding synchronization facility on the path of the request flow from the client to the DKVS and to the history server by having the DKVS and the history log strictly honours the timing of clients' request issuing. However, such synchronous nature of previous solutions is limited due to its strict serialization applied to order operations unnecessarily for DKVSs and a history log.

4. Materials and Methods

In this section, we discuss the architecture of the proposed CSS named Relief that employs novel methods to solve aforementioned limiataions of existing solutions. We

start with explaining the overall architecture of the system, followed by the description of the interface which Relief provides for clients. Then, we present its core mechanism; a conflict-free replicated history update. Subsequently, the format of request and response messages is specified. Furthermore, the description of the consistency verification using the history provisioned by Relief is discussed next.

4.1. Relief Architecture

The architecture of Relief is depicted in Figure 7. Relief clients are running on user's end devices and send requests to cloud servers and receive responses. *Relief controller* runs on a cloud server. Relief controller handles requests received from clients and sends back responses to clients accordingly. It is the core component of Relief that mediates access to their data and the history stored in a DKVS. Relief controller interacts with *data manager* to read or write data from the DKVS and interacts with *history manager* to update or read the history from the DKVS. Both data manager and history manager directly interact with the DKVS to access data and the history, respectively. We discuss more details of how Relief controllers mediate access to data and the history by interacting with the data manager and the history manager in Section 4.2.

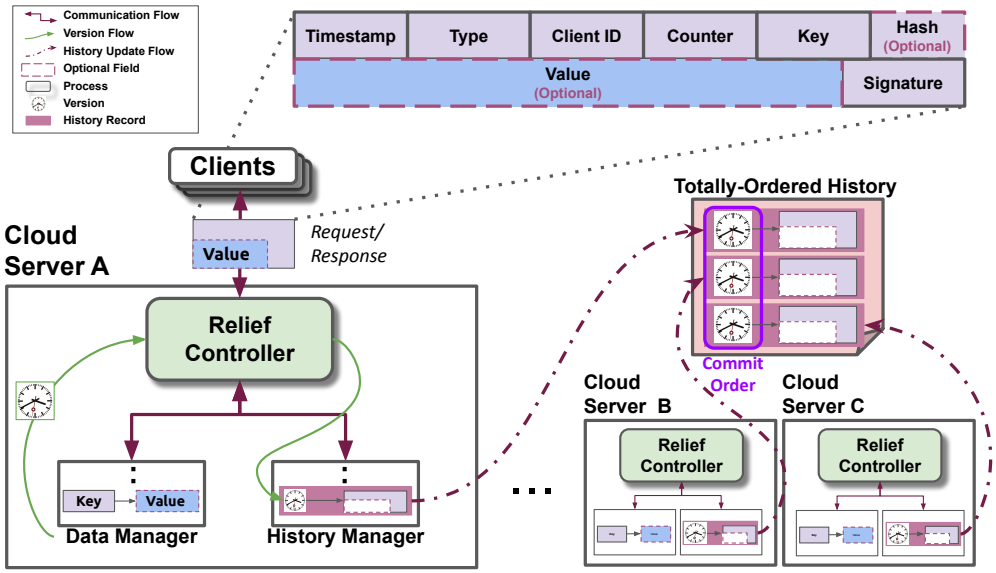

Figure 7. Relief Architecture.

For the specification of the DKVS that Relief can be integrated with, we require only two properties. One property is related to committing operations on data which is that the DKVS must expose versions determining the total order of operations. The other required property is related to the history update is that the DKVS must allow eventual consistency for updating the history. Nothing blocks us from using two different DKVSs for storing data and the history as long as each DKVS satisfies those requirements correspondingly.

Relief controllers expose the RPC interface for clients to invoke to send requests and receive responses on the return. Essentially, Relief supports the similar interface as Caelus. Below is the details of those relevant to a history.

1. `GET(k)v`: Reading a value for the key k. It is expected to return a value v. If the key k does not exist, then *nil* is returned for the value v.
2. `PUT(k,v)`: Writing a value v to the key k.
3. `READ_HISTORY(s,e)H`: Reading a section of the history between the start version s and the end version e where $s \leq e$. It returns H, a list of entries sorted based on the version. All entries in H are indexed by versions falling in the range between s and e. If the given e is larger than the version of the latest entry in the history, then the history returns the list of entries up to the latest entry.

4. `WRITE_ATTEST(s,e)`: Writing an attestation that attests the section of the history between the start version s and the end version e.
5. `READ_ATTEST(s,e)` <A,H,s',e'>: Reading attestations that attest the section of the history between the start version s and the end version e. Along with A, the list of attestations, H, the section of the history is also read and returned similar to `READ_HISTORY(s,e)`. However, if there are attestations that attest the section between s and e and the outer range, then the entire section of the history attested by those attestations is inclusively retrieved. The response also includes the start version s' and e' to precisely specify the section of the history and attestations retrieved.

4.2. Conflict-Free Replicated History Update (CRHU)

We devised a technique called *conflict-free replicated history update* (CRHU) to help CSSs provide eventually consistent totally ordered history, while keeping the history server scalable and highly available. The technique is based on our observation that *the underlying DKVS already has the means to determine the order of operations it commits and can create a total order using clocks*. Section 2.3 discusses various usage of clocks representing logical or physical time to determine ordering between distributed events. Existing DKVSs utilize those clocks which are used to eventually create a total order of committed operations. Here, we clearly define that the timestamps of clocks that reflect partial order (or total order as the special case of a partial order) assigned by the underlying DKVS as *version*. CRHU works by having the underlying DKVS expose the version of each operation committed in the data keyspace. Thus, one important assumption we make about the underlying DKVS is that versions must be exposed, and those versions should reflect the actual *commit order*. When the Relief controller receives a response from the data keyspace, it takes the version for the requested operation and uses that version as a key to store a value, which is the meta-data of the requested operation, as illustrated in Figure 7. Subsequently, Relief controller writes the key-value pair in the history keyspace under eventual consistency.

As time goes by, writes to the history keyspace will eventually be fully replicated, and the totally ordered history visible to everyone keeps growing. Although replication occurs asynchronously under eventual consistency, there is no conflict for the history updates made to the history keyspace because the version used as the key is what the DKVS has already used to totally order committed operations. Therefore, the history is constructed eventually in a conflict-free way. Assuming there is a time-bound for how fast the DKVS replicates, we can estimate the expected time taken for clients and attestors to see the history supposed to reflect the ordering of all operations committed.

Failures may certainly occur on any Relief node, and replication can be interrupted. Such interruption may last longer than expected time-bound. Those failures are all revealed to Relief clients, as those anomaly results into the omission of operations. Such transparency gives clients the opportunities to localize faults and pick different nodes for subsequent accesses. In addition, CSS providers may also utilize this feature to monitor the health of their services and closely track which nodes are more likely to cause failures so that they can set up more timely contingency plans and countermeasures.

CRHU piggy-backs on already existing and actively used mechanisms of DKVSs not only to figure out *the total order of committed operations* in the data keyspace but also to *speculatively constructs the totally ordered history reflecting the actual commit order of operations* for the history keyspace. Accordingly, CRHU allows Relief to create a totally ordered history containing the set of operations requested by clients and indexed by versions reflecting the commit order assigned by the underlying DKVS.

Algorithm 1 shows how a Relief controller handles client requests. Initially, *dataHandle* and *historyHandle* are obtained by establishing connection to the underlying DKVS. Both handles must be accessing separate keyspaces so that they do not interfere each other's operations. Then, the controller starts the loop where each iteration starts with a received request from a client and terminates when the client disconnects. This loop may execute in parallel by multiple threads at the same time.

Algorithm 1: Relief Controller's Request Handling Algorithm

```
dataHandle ← dataManager.connect();
historyHandle ← historyManager.connect();
r_i ← recv();
while r_i ≠ null do
    if r_i.type is Put then
        k_i ← r_i.getKey();
        v_i ← r_i.getValue();
        result_i ← dataHandle.put(k_i, v_i);
    else
        k_i ← r_i.getKey();
        result_i ← dataHandle.get(k_i);
    end
    if result_i.ack is True then
        t_i ← result_i.version;
        historyHandle.put(t_i, r_i);
    end
    send(result_i);
end
```

The first thing the controller does is checking the type of the request. If the request is a Put request for writing data, the controller gets key and value of the Put requests from its meta-data fields. Then, the controller actually performs the Put operation by invoking data manager's function. Data manager's put function is the wrapper of the underlying DKVS API call which is slightly instrumented to return the result along with versions the DKVS internally uses to determine the total order of operations. Get path is same as the Put path previously described except that it only needs to use key from the client's request's meta-data field and invoke a get API call of the data manager. Again, the data manager's get interface is the wrapper of the DKVS's get API calls except for expecting the result including version.

Sequentially, the controller checks if data manager succeeded to perform the requested operations. If the result was acknowledged which can be checked by seeing if the value of the result's acknowledgement is True, then the controller retrieves the version returned and perform put operation of the history manager. The put operation of the history manager is using eventual consistency unlike data manager's counterpart which can use an arbitrary consistency model as requested by the client. If the results' acknowledgement is not True, then the controller does not invoke the put interface call to the history manager not to update the history with failed operations. Finally, the controller sends the result from the data manager back to the client.

An example of how CHRU works is illustrated in Figure 8. Initially, Client A and Client B concurrently issue `Put` requests under linearizability. Controllers on Node A and Node B perform `Puts` to their data keyspace. DKVS determines the order of those concurrent `Puts` to be *Put(x,1)* and then *Put(x,2)*. DKVS responds with versions for those operations that are t1 and t2, respectively. Controllers respond to clients and, at the same time, perform `Puts` in parallel to update the history keyspace with the new operations *Put(x,1)* and *Put(x,2)*. The history replica on the Node A contains only *t1*→ *Put(x,1)*, while the history replica on the Node B contains only *t2*→ *Put(x,2)*. Then, after *visibility time* past, we expect updates to the history keyspace to be fully replicated, and both history replicas merge those updates with no conflict. Thus, the history replicas eventually contain both *t1*→ *Put(x,1)* and *t2*→ *Put(x,2)*. They are ordered using the versions in the figure, but the data placement is managed by DKVS. Subsequently, clients concurrently issue `Get` and `Put` requests under eventual consistency. The controller on Node A performs *Put(x,3)* to its data keyspace, while the controller on Node B performs *Get(x)* to its data keyspace. Consequently, DKVS processes *Get(x)* at t3 before *Put(x,3)* at t4. The controller on Node A

receives the DKVS's response with the version t3 which is when *Get(x)* was committed and the value 2 which was written by *Put(x,2)* at t2. The controller on Node B simultaneously receives the DKVS's response with version t4. Both controllers respond to their clients and concurrently issue `Puts` to their history keyspace to update their history replicas with those operations just committed by DKVS. Accordingly, the history replica on Node A contains *t1→ Put(x,1)*, *t2→ Put(x,2)* and *t4→ Put(x,3)*, while the history replica on Node B contains *t1→ Put(x,1)*, *t2→ Put(x,2)* and *t3→ Get(x)2*. Those updates are asynchronously replicated and merged with no conflict and the both history replicas contain *t1→ Put(x,1)*, *t2→ Put(x,2)*, *t3→ Get(x)2* and *t4→ Put(x,3)*. Thus, the history is totally ordered and honors the actual commit order assigned by DKVS. Note that the updates to the history replicas have been made concurrently and asynchronously.

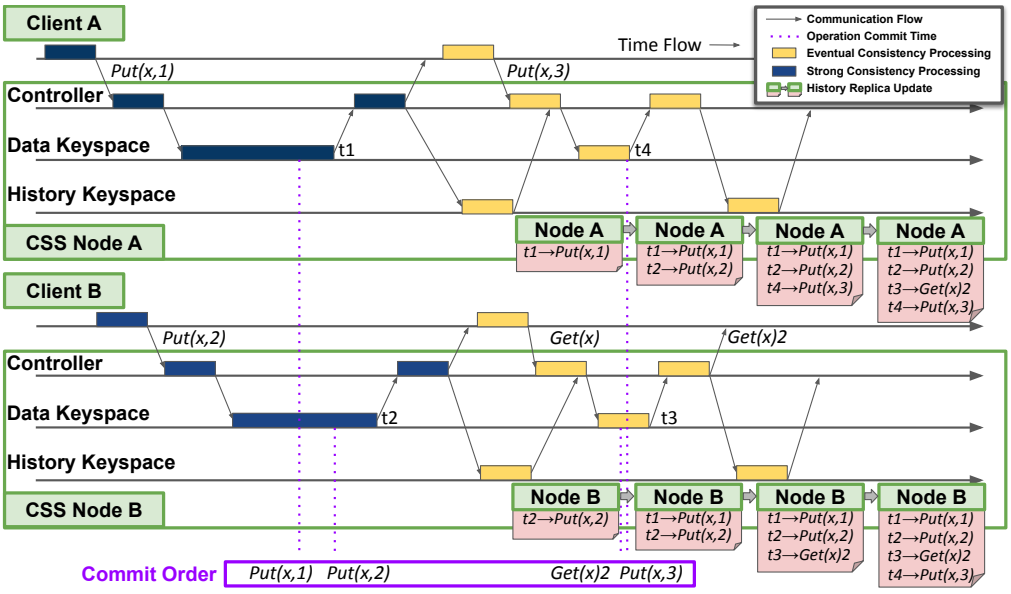

Figure 8. An Example Execution of Conflict-free History Replica Update. This figure shows how clients' requests for operations are processed on CSS Nodes, recorded in the distributed history replicas in parallel and eventually merged into a totally ordered history reflecting commit order of operations. Also, note that history updates are executed in parallel and asynchronously by using eventually consistenty `Puts` to update the distributed history replicas.

For a `READ_HISTORY` request, a controller performs a ranged query to the history keyspace. The start and the end of the range are given by clients in the `Value` field of the request as described in Section 4.3. The ranged query uses those versions as parameters and retrieves all history entries by reading all key-value pairs in the history keyspace whose key is falling between the range specified by given parameters.

Because the history update is eventually consistent, there may be operations logged in one history replica but not in other history replicas. There are two possibilities such a situation can occur. First, the replication is not yet completed. In this case, once the replication completes, all history replicas will contain the same set of entries in the same order. Second, there is a failure that occurred on the cloud. The source of failures may vary: the crash of CSS nodes, disconnection of network links or the delay due to a heavy load. In the second case, clients will see the history that does not contain all operations transparently and may take appropriate actions to cope with the faulty nodes. Caelus merely has nodes halt and raise an alarm as soon as missing operations in the history log are detected, because it thinks that the visibility timebound assumption breaks only if malicious attacks has made the operation not visible. However, we let Relief expose server-side failures to clients, even if it is not caused by a malicious attack. Clients can determine whether to take an action or gracefully ignore some degree of inconsistencies.

4.3. Request/Response Format

Relief supports the same interface as Caelus's. Relief's clients prepare their requests accordingly by filling out fields formatted as described in Figure 7. The more detailed description of each field is given in Table 1. For any request, the field `Timestamp` is assigned with a timestamp obtained from a client's clock synchronized via the well-known NTP clock synchronization protocol. The field `Type` specifies which type of operation is requested, as there are several operation types required by Caelus, such as `Get`, `Put`, `Read_History`, `Write_Attest`, `Read_Attest`, `Select`, `Wake` and `Status`. The client sending a request is identified by the field `Client ID`. Each client should be initially associated with a UUID as their identifier specifying who sent a request. To fill the field `Counter`, a client uses a monotonically increasing sequence number as a counter value for each request issued to keep track of causality between its operations during consistency verification. The field `Key` specifies a key a client reads or writes. The size of field `Key` depends on the maximum key size supported by the underlying DKVS. The field `Value` contains the actual data content to write or read for Puts and Gets, respectively. The size of the value field is varied depending on the maximum size allowed by the underlying DKVS. For our prototype, we built on top of DynamoDB as our backend portion of the DKVS. The table shows the maximum size of key and value sizes based on DynamoDB, but it can be flexible to change if a different backend DKVS is chosen. The field `Hash` contains the SHA-512 hash sum of the field Value. The field `Hash` is required to prevent malicious attackers from tampering with data in the field `Value`. The field `Signature` contains the digital signature of the request based on a 3072-bit RSA Digital Signature. The field `Signature` includes every meta-data field, including the field `Value` to detect forging, tampering, or equivocating attacks against requests and responses. We separately store the field `Value` in a data keyspace and all other meta-data fields except for `Value` in a history keyspace. The entire meta-data field is small enough for most DKVSs as it requires $349 + |Key|$ Bytes.

Table 1. CSS Request and Response Format Description.

Field	Size	Request	Response
`Timestamp`	64 bits	Timestamp assigned by a client.	For GET, it is the matching PUT's timestamp.
`Type`	8 bits	The specifier for several operation types in Caelus.	For GET, it is the matching PUT's type.
`Client ID`	128 bits	The UUID of a client	For GET, it is the matching PUT's client ID.
`Counter`	32 bits	The monotonically increasing sequence number for each request per client.	For GET, it is the matching PUT's counter.
`Key`	2048 bytes	The key of values either to read or write.	For GET, it is the matching PUT's key.
`Value`	400 KB	Value to write for PUT. It is null for GET.	Value to read for GET. It is null for PUT.
`Hash`	512 bits	The SHA-512 hash the field Value. For GET, it is null.	The hash of the value reading for GET. It is null for PUT.
`Signature`	3072 bits	The RSA signature of the all meta-data fields, generated by the requesting client.	The RSA signature of the all meta-data fields, generated by the responding CSS.

We use the same format for both requests and responses for the simplicity of the interface, but an alternative format for responses can be used to save network bandwidth consumption. Each field may have a different meaning for responses depending on both the type of the operation. If it is a response for a `Get` request, `Timestamp`, `Type`, `Client ID`, `Counter` and `Key` are used to uniquely specify the `Put` operation from which `Get` reads

`Value`. Also, the field `Value` can be used to store not only actual data a client reads and writes but also protocol-related data. For a response to a `Put` request, the field `Value` has no meaning. Therefore it is just null. However, for a response to a `Get` request, the field `Value` contains the actual data a client reads. Moreover, the field `Hash` is used for a response to a `Get` request to store the hash sum of the data in transit. Trivially, its purpose is to confirm that data in transit has not been tampered during the network transmission. However, the filed `Hash` has no meaning for a response to a `Put` request, therefore set to null. The signature of a CSS is used for the field `Signature` for every type of responses to remove the possibility of CSS's equivocation.

Furthermore, there are special cases where we use a meta-data fields with different meaning for the protocol-specific operations that are not `Get` or `Put`. For a `Write_Attest` request, because the request does not need to write any value, we use the field `Hash` to store the hash sum of the history segment for which the attestor is attesting. When we use the field `Hash` for a `Put` request, we store the hash sum of actual data in `Value` to prevent tampering attacks. `Get_History` and `Read_Attest` requests do not use the field `Hash`. A `Get_History` request uses the field `Value` to store the start version and the end version for the CSS to issue a range query to the DKVS to read a segment of history from the history keyspace. Similarly, a `Read_Attest` uses the field `Value` in the same way but to read attestations from the data keyspace. A `Write_Attest` request fills the field `Value` with the actual attestation that is going to be stored in the data keyspace. Keys used to store attestations are derived from the start and end versions and allocated as special keys only for protocol-specific operations. In addition, clients can send `Select` and `Status` request [13]. For a `Select` request, the root attestor client uses the field `Key` to write to the specially allocated key dedicated for selecting a new active attestor for the new root attestation period. Also, it uses the field `Value` to designate the specific client ID of the active attestor for the following root attestation period. For a `Status` request, clients write to specially allocated keys, which fills up the field `Key`, dedicated to storing each registered client's device status update, which fills up the field `Value`.

4.4. Data Consistency Verification

Relief exposes a history to clients for them to verify the consistent behavior of the underlying cloud storage service. Relief's consistency verification can be viewed as a white-box approach as its CSS reveals internal information regarding the actual commit order of operations in the underlying DKVS. With the history, consistency verification is deterministic as opposed to previous approaches using black-box approaches [3,4], which require exploring every interleaving of concurrent operations because the actual commit order is not exposed to clients.

Relief is integrated with a consistency oracle [26]. For a given history exposed by a CSS, Relief injects each operation into the consistency oracle. Furthermore, for every read operation, it additionally throws a query to the oracle to obtain the list of valid values. Then, Relief checks if the actual value returned for the read operation was included in the list of valid values to read returned by the consistency oracle. Relief's approach is more efficient than black-box consistency verification approaches. With the black-box approach, serializability is a well-known NP-complete problem, while Relief takes one linear search through the log.

5. Results

5.1. Evaluation Setup

We implemented a prototype CSS and our client is integrated with YCSB (Yahoo Cloud Serving Benchmark) framework [27]. We simulated a DKVS exposing versions by obtaining timestamps from physical clocks when the DKVS commits operations. We used the timestamps as the approximation for versions. This simulates a DKVS that uses physical clock to determine the order of operations.

With this prototype, we both evaluated the server's performance and consistency verification speed. First, the performance of Relief was measured on our lab facility consisting of three physical machines connected over 1Gbps Ethernet. The underlying DKVS was configured to use eventual consistency. Client threads (YCSB) ran on a machine equipped with a Intel Xeon CPU E5-2698 v4 which has twenty 2.20 GHz cores and 504 GB RAM. We picked this machine to generate concurrent client operations using many CPU cores. On the other hand, two Relief servers ran on two identical physical machines, each of which is equipped with an Intel Core i5 CPU with two 3.20 GHz cores and 8 GB RAM. One was the primary, and the other was set to be the secondary. With this setting, both throughput and latency were measured for three different modes. As a baseline, history logging was not used. Thus, there was no performance overhead involved in history logging. To measure the performance of previous approaches, we used sequentially consistent history as used by Caelus. In this setting, every history logging request was sent to the primary, and the primary serializes it. Then, we measured the performance numbers for Relief's eventually consistent history logging.

Second, using a machine equipped with Intel Core2 Quad CPU Q9550, which has 2.83 GHz cores and 8 GB RAM, we measured the speed of client-side data consistency verification for strong consistency models, given the history exposed by Relief. We ran a single YCSB process configured to run with 8 threads where the generated I/O follows Zipfian distribution. Then, we read history from the Relief server and performed the consistency verification for strong consistency. We varied the percentage of read operations and measured the time taken for verification first. Then, we varied the number of operations as well as the percentage of read operations and measured the time taken for verification.

5.2. Performance

To measure the throughput of three different CSS architectures mentioned earlier, we used the workload consisting of 1000 operations for 1000 pre-populated records. The percentages of read and write operations was 50% each, and the workload followed the Zipfian distribution. We ran two instances of YCSB workload: one for the primary server and the other for the secondary server. We varied the number of threads for each YCSB workload from 2 to 128. We measured the throughput for the baseline with no history logging, sequentially consistent history, and eventually consistent history that was constructed using CRHU.

Figure 9 shows the result of our throughput measurement experiment. We expected that Relief could handle as much throughput as the baseline because Relief uses eventual consistency for history updates as well as data access. Eventual consistency allows Relief servers to read from and write to the local storage, which is basically the same as the baseline except for negligible overheads involved in extra computation regarding history updates. As expected, our results show that Relief's throughput was on par with the baseline. Also, we note that throughput improvement was doubled by Relief compared to Caelus. This is because Caelus uses sequential consistency for history updates and operations are pending until history updates are serialized and committed by the primary. Unlike Caelus, Relief can take full advantage of having two servers as it employs eventual consistency. The performance increased linearly as we threw in more cores. As more threads were used to generate more I/O bursts, the throughput gap between Relief and Caelus became wider. For instance, using 128 threads, Relief's throughput was higher than Caelus by about 110 operations per second, while when using 2 threads, the difference was only about 2 operations per second. This shows that under the heavy workload, Relief can significantly outperform Caelus in terms of the throughput.

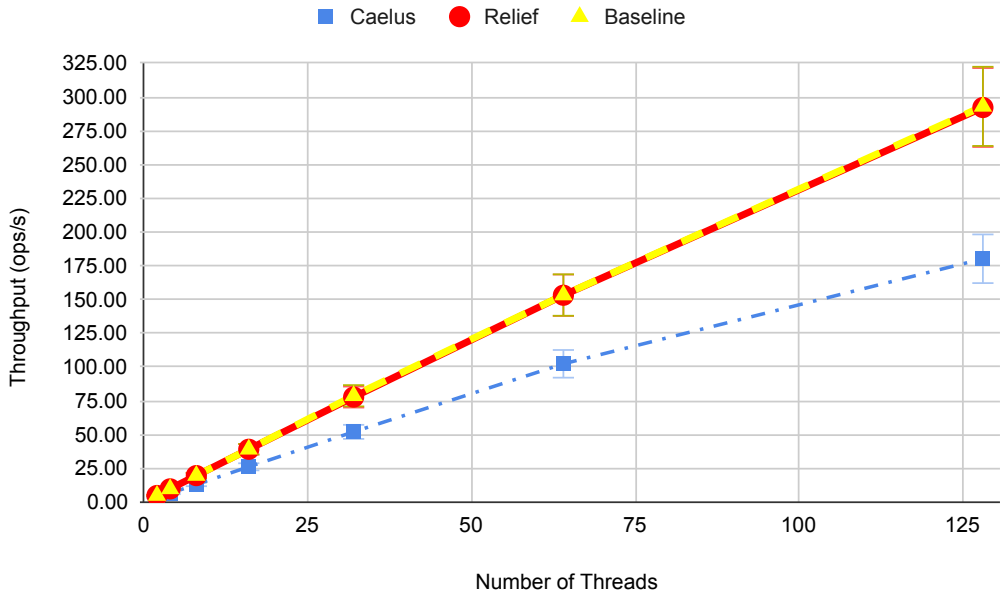

Figure 9. Relief throughput chart.

To confirm our reasoning about the throughput difference between Relief and Caelus, we collected the throughput of the primary and the secondary separately. Figure 10 shows the throughput for the primary server, while Figure 11 shows the throughput for the secondary server. For the primary server, there was little difference between Relief and Caelus except that Caelus's throughput for the primary server was slightly lower than the baseline for 128 threads. This is because the primary server starts getting affected by the history updates from the secondary server. Unlike the primary server, the secondary server shows much more significant throughput degradation. The secondary server's throughput for Caelus was only 33.3% of that of Relief with 128 threads. As we explained earlier, this is due to the cost of having operations wait for coordination regarding history updates.

We also measured the average latency for client operations—the latency includes the time taken to update the history. The result is presented in Figure 12. We collected the average latency for read operations and write operations separately. Also, we gathered the average latency for the primary server and the secondary server independently as well. Regardless of the type of operations, it took about 3× higher average latency for operations sent to the secondary server under Caelus mode, compared to Relief or the baseline. Again, this is because those operations were delayed until the primary server completed synchronous updates to the history. Thus, we confirm that using sequentially consistent history updates can directly significantly affect the throughput and the average latency. On the other hand, the average latency for the primary server remained the same for Caelus, Relief and the baseline.

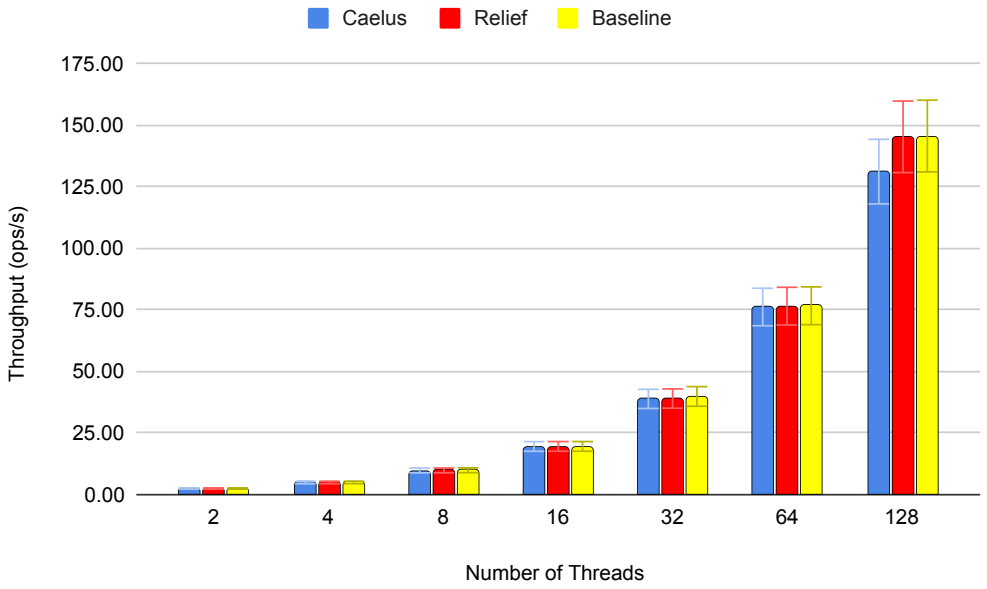

Figure 10. Primary server throughput.

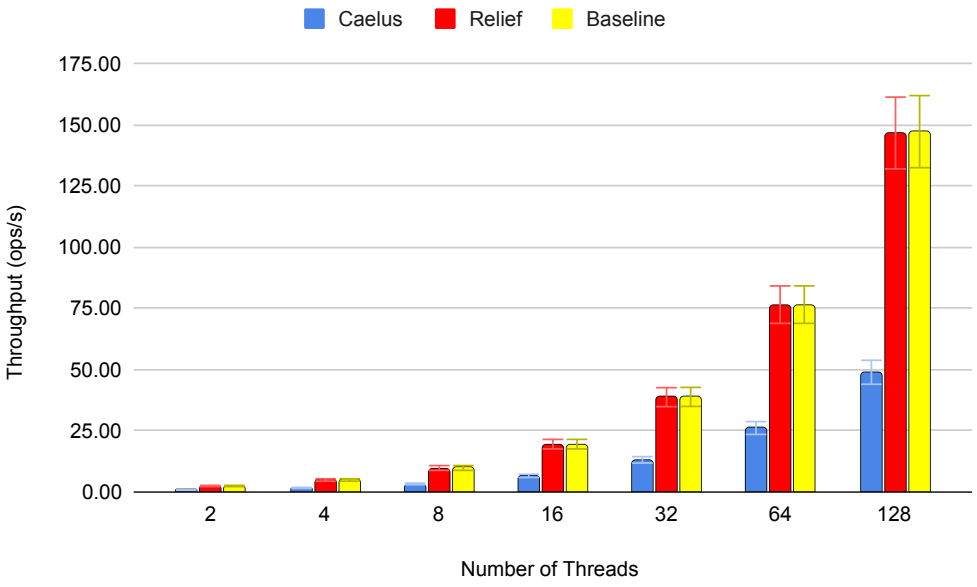

Figure 11. Secondary server throughput.

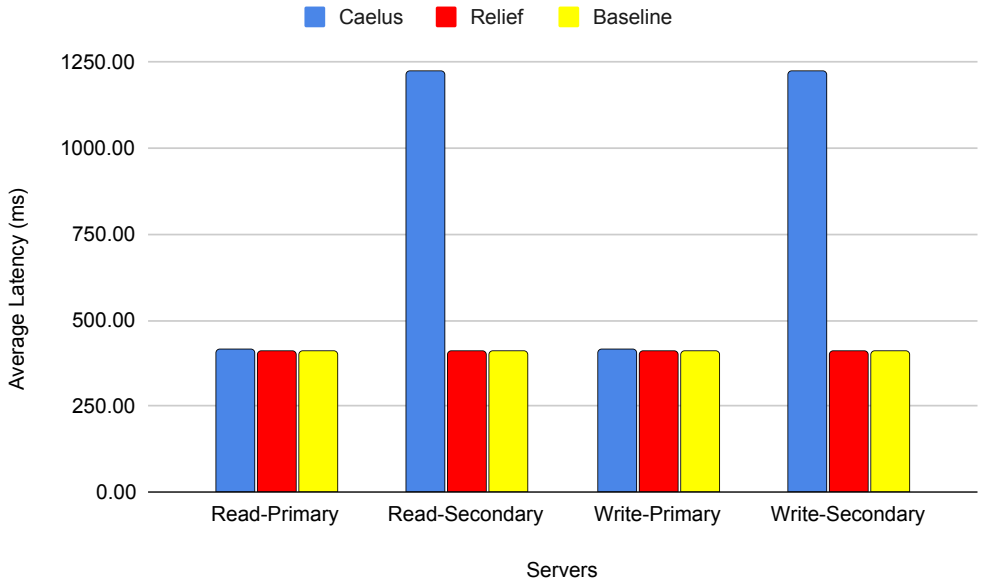

Figure 12. Average Latency.

5.3. Verification Speed

Relief exposes a log containing a history of operations that is totally ordered. Clients can know the exact order and timing of committing operations using the log because all committed operations are explicitly ordered based on versions assigned by the underlying DKVS. With this, consistency verification only requires a simple linear search to iterate over each entry in the log to analyze the history. This makes consistency verification highly efficient compared to black-box consistency verification approaches used by Jepsen, the state-of-the-art database consistency fuzzer. Because black-box verification explores all possible interleaving of concurrent operations to check if the given history is valid under specific consistency models, it is essentially an intractable combinatorial explosion problem. For example, as mentioned earlier, serializability verification is known to be an NP-complete problem. Unlike previous black-box approaches, Relief's verification is a polynomial-time algorithm, and therefore, it is inherently much faster.

We fixed the number of operations to be 1000 for 100 records but varied the proportion of read operations from 0 to 100% of all operations. Read operation verification involves checking if the value returned is the latest. Therefore it slows down the verification compared to write operation verification. How fast Relief could verify the consistency of operations under this setting is shown in Figure 13. We measured the average latency of verification per operation for various read operation percentages of YCSB workloads. Verification of 1K operations always remained below 1 ms. In addition, as a higher percentage of read operations was generated, the verification became slower.

Subsequently, we fixed the percentage of read operations and varied the number of operations from 500 to 5000. In addition, we repeated with varying percentages of read operations. We present the result of the measurement in Figure 14, which shows how cumulative time was taken for verification with each setting. As more operations were verified, the latency grew linearly, unlike black-box consistency verification approaches, where the cumulative time taken for verification grew exponentially as the number of operations increases. As we mentioned earlier, we have observed the tendency to increase cumulative verification time for the increased percentage of read operations. Thus, this confirms that more validity checks are required as the percentage of read operations grows.

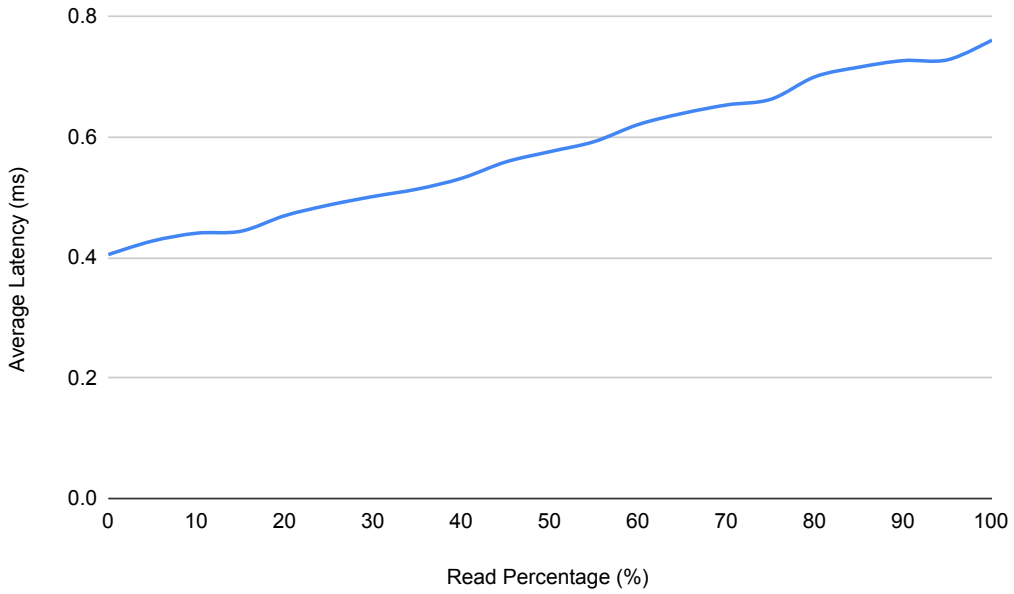

Figure 13. Average Verification Latency per Operation for Various Read Percentages.

Figure 14. Cumulative Time taken for Consistency Verification.

6. Discussion

Our empirical study shows that Relief cloud storage service architecture improved throughput by 2 times compared to the previous architecture using sequentially consistent history server component. In addition, due to its eventual nature in terms of updating the history log, workloads could be scaled well and therefore we could shorten read latency by three times compared to a strongly consistent approach. Moreover, the verification speed with respect to consistency check, we demonstrate that the latency only gets linearly grows as reading percentages and operations increase.

Relief's architecture supporting eventual consistency for history updates improves the previously existing consistency verification service architecture in terms of both throughput and latency. These improvements imply that it is practical to support white-box consistency verification as opposed to black-box consistency verification used by most previous proposals. Relief enables this by having DKVSs expose the version they internally use to

determine the order of operations for history updates for consistency verification of clients. In practice, we believe most DKVSs can be slightly instrumented and easily integrated with Relief to employ the CRHU to enable white-box consistency verification. Hence, Relief's contribution is making deterministic white-box consistency verification more widely adaptable to various cloud storage services at the cost of a slight modification to their underlying DKVSs and addition of the history logging layer. In the future, we would perform more empirical studies of Relief by integrating with various popular DKVSs existing today.

6.1. Related Work

The most closely related works to Caelus are SUNDR [5], BFT2F [10], and CloudProof [11]. These systems provide consistency and integrity guarantees for untrusted storage systems to clients who do not communicate directly with each other. SUNDR only guarantees fork consistency, while BFT2F weakens fork consistency to fork*. Other work has also extending SUNDR's contribution on fork-linearizability to computations on untrusted services [28,29]. Both fork, fork*, and fork-linearizability are weaker than any of the consistency models that Caelus can guarantee in that they permit some operations to be forever unknown to some clients. CloudProof can verify strong consistency but requires information from clients to be assembled at an "auditor". Because the auditor is not always online, auditing is retroactive instead of in real-time. Caelus uses a smartphone to make auditing real-time and distributes the auditing work to minimize the impact on the smartphone battery.

Depot [6], SPORC [7] and Venus [8] provide consistency guarantees using client-to-client communication. Client-to-client communication simplifies the problem because clients may implement their replication policy and thus enforce a consistency model independent of the cloud service provider. However, client-to-client communication is inefficient for battery-powered devices. Also, it requires a trusted service that can buffer and multicast messages so that clients need not waste battery on network bandwidth or not unnecessarily stay awake simultaneously to communicate. Caelus avoids these by devising a protocol that can use the cloud service to buffer and multicast messages without trusting the cloud service.

Timeweave [30] was an early use of attested histories to verify the actions of an untrusted party. Since then, the idea of using an attested history has been applied to detect misbehaving virtual machines [31], misbehaving replicas in BFT systems [32,33], as well as to improve the performance of BFT systems [34,35]. Recent work has also proposed the use of trusted platform modules (TPMs) as integrity verifiers for cloud infrastructure [36]. However, none of these previous works directly address the problem of consistency verification.

Several cryptographic file systems also guarantee freshness [37–42]. However, they all assume that all operations are linearizable so that they only need to check that the latest values are read by a client (i.e., strong consistency). As stated in Brewer's well-known CAP theorem, systems that enforce strong consistency cannot scale. In contrast, Caelus protects systems with weaker consistency models such as eventual and causal consistency, which are more suitable for globally distributed cloud infrastructure.

Finally, other work has proposed distributing data across multiple cloud services to protect the integrity and recoverability of data [43–45], as well as using cryptographic techniques to probabilistically prove retrievability [46,47], data possession [48], or whether data is encrypted properly [49]. However, these systems do not address data consistency and, in the case of the cryptographic techniques, mostly assume static data. On the other hand, Caelus does not directly address recoverability or retrievability, combining these techniques with Caelus interesting future work.

6.2. Cloud Storage Service Architecture Comparison

As mentioned earlier, several research projects have proposed a cloud storage service architecture to enable data consistency verification. In this section, we provide a

direct comparison between our solution in this work and those previous related works. Table 2 summarizes the comparison analysis. The first column shows the name of the work, the second column specifies whether there is a history server component, the third column describes consistency models that can be verified with the corresponding CSS architecture, the fourth column specifies whether verification method is deterministic or non-deterministic, and the fifth column shows if history update is serialized or not.

Table 2. CSS Architecture Comparison.

Name	History Server	Consistency Models	Verification Method	History Update
SUNDR	Yes	Strong Consistency	Deterministic	Serialized
CloudProof	Yes	Strong Consistency	Deterministic	Serialized
Depot	No	Fork-Join-Causal-Consistency	Deterministic	Not serialized
Jepsen	No	Multiple	Non-deterministic	Not serialized
Cobra	No	Serializability	Non-deterministic	Not serialized
Caelus	Yes	Multiple	Deterministic	Serialized
Relief	Yes	Multiple	Deterministic	Not serialized

The work with a history server component can allow clients to be battery-friendly and not to require direct communication channel establishment between clients, because clients do not need to be online all the time nor exchange views by themselves. View exchange is done via the history server and therefore it can occur asynchronously.

Related works can be also divided in two types. Those ones that can support verification of multiple consistency models or not. Previous works that use the log of operations as the form of commit history can support multiple consistency verifications because they are not bound to a specific consistency model due to cryptographic protocols constructing the history of operations.

Verification method can be viewed as deterministic if it scans each operation and can perform analysis without sorting or recursive search through the history. Deterministic verification is computationally efficient, while non-deterministic verification can be intractable because there can be too many interleaving of concurrent operations to sort and recursively search through. Cobra [3] and Jepsen [4] are enumerate every possible interleaving of concurrent operations, which can face with exponential growth in the number of possible interleaving making the verification computationally intractable. Unlike those systems using such a non-deterministic verification checks inherently perform worse than deterministic methods like ours which requires just one pass of linear iteration over entries in the history log.

History update may be serialized or not. Serialized history update usually involves an agent that imposes an order to each operation across views of different clients. While doing so, a specific total order can be determined arbitrarily. Yet, serialized history update may suffer from the scalability problem because the update must be processed by the agent imposing the order.

As one can see from the Table 2, Relief is the only cloud storage service architecture that provides a history server, multiple consistency model verification, deterministic verification support, and non-serialized history updates.

7. Conclusions

Enabling consistency verification for clients is becoming a more desirable feature as many cloud storage services provide various consistency models. Using versions DKVSs

internally employ to determine the actual commit order of operations, eventually consistent totally ordered history can be provided for clients to perform consistency verification. We show that throughput and latency can be significantly improved compared to previous approaches relying on strong consistency. Moreover, with such a log, consistency verification can be performed in a white-box manner by scanning through the log linearly once. As a result, we believe that the proposed architecture can support easier adoption of various consistency models by relieving the concern regarding data consistency.

Limitation and Future Work

Relief uses versions that are provided by the underlying DKVS. Currently, we are not aware of DKVSs exposing versions on the response to client requests. ZooKeeper uses 'zxid' sequence numbers and returns those numbers for client requests, but those numbers are used to determine the order among writes while only relative orders of reads with respect to writes can be inferred. In the future, we will explore various DKVS to broaden our understanding of the exposure of versions. Another limitation of Relief is that there is no support for transactions. The transaction is regaining attention as a key feature of ACID required for many DKVS applications. Although Relief considers the granularity of each read and write for consistency checking, we will explore a way of adding transaction supports for data consistency verification in the future.

Author Contributions: Conceptualization, B.-H.K. and Y.Y.; methodology, B.-H.K. and Y.Y.; software, B.-H.K.; validation, B.-H.K.; formal analysis, B.-H.K.; investigation, B.-H.K.; resources, B.-H.K. and Y.Y.; data curation, B.-H.K.; writing—original draft preparation, B.-H.K. and Y.Y.; writing—review and editing, B.-H.K. and Y.Y.; visualization, B.-H.K.; supervision, Y.Y.; project administration, B.-H.K. and Y.Y.; funding acquisition, Y.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the Ministry of Trade, Industry & Energy (MOTIE) and the Korea Institute for Advancement of Technology (KIAT), under Grants P0014268 Smart HVAC demonstration support, by the Basic Science Research Programs through the National Research Foundation of Korea (NRF) funded by the Korea government (MSIT) (2020R1F1A104826411), by Korea Agency for Infrastructure Technology Advancement (KAIA) grant funded by the Ministry of Land, Infrastructure, and Transport (Grant 21TLRP-B148676-04), and by the 2021 Hongik University Research Fund.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; nor in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

CSS	Cloud Storage Service
DKVS	Distributed Key-Value Store
CRHU	Conflict-free Replicated History Update

References

1. Research and Markets. Global Cloud Storage Market (2020 to 2025)—Rise of Containerization Presents Opportunities. 2020. Available online: https://www.globenewswire.com/news-release/2020/10/16/2109759/0/en/Global-Cloud-Storage-Market-2020-to-2025-Rise-of-Containerization-Presents-Opportunities.html (accessed on 25 March 2021).
2. Terry, D. Replicated data consistency explained through baseball. *Commun. ACM* **2013**, *56*, 82–89. [CrossRef]
3. Tan, C.; Zhao, C.; Mu, S.; Walfish, M. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, 4–6 November 2020; pp. 63–80.
4. Kingsbury, K. Distributed Systems Safety Research. 2016. Available online: https://jepsen.io/ (accessed on 21 March 2021).
5. Li, J.; Krohn, M.N.; Mazières, D.; Shasha, D.E. Secure Untrusted Data Repository (SUNDR). In Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, CA, USA, 6–8 December 2004; pp. 121–136.

6. Mahajan, P.; Setty, S.T.V.; Lee, S.; Clement, A.; Alvisi, L.; Dahlin, M.; Walfish, M. Depot: Cloud Storage with Minimal Trust. In Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, Vancouver, BC, Canada, 4–6 October 2010; pp. 307–322.
7. Feldman, A.J.; Zeller, W.P.; Freedman, M.J.; Felten, E.W. SPORC: Group Collaboration using Untrusted Cloud Resources. In Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, Vancouver, BC, Canada, 4–6 October 2010; pp. 337–350.
8. Shraer, A.; Cachin, C.; Cidon, A.; Keidar, I.; Michalevsky, Y.; Shaket, D. Venus: Verification for untrusted cloud storage. In Proceedings of the 2nd ACM Cloud Computing Security Workshop, CCSW 2010, Chicago, IL, USA, 8 October 2010; pp. 19–30.
9. Kim, B.H.; Huang, W.; Lie, D. Unity: Secure and durable personal cloud storage. In Proceedings of the 2012 ACM Workshop on Cloud computing security, CCSW 2012, Raleigh, NC, USA, 19 October 2012; pp. 31–36.
10. Li, J.; Mazières, D. Beyond One-Third Faulty Replicas in Byzantine Fault Tolerant Systems. In Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI 2007), Cambridge, MA, USA, 11–13 April 2007.
11. Popa, R.A.; Lorch, J.R.; Molnar, D.; Wang, H.J.; Zhuang, L. Enabling Security in Cloud Storage SLAs with CloudProof. In Proceedings of the 2011 USENIX Annual Technical Conference, Portland, OR, USA, 15–17 June 2011.
12. Tomescu, A.; Devadas, S. Catena: Efficient Non-equivocation via Bitcoin. In Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2017; IEEE Computer Society: Los Alamitos, CA, USA, 2017; pp. 393–409. [CrossRef]
13. Kim, B.H.; Lie, D. Caelus: Verifying the Consistency of Cloud Services with Battery-Powered Devices. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, 17–21 May 2015; pp. 880–896.
14. Li, K.; Tang, Y.R.; Kim, B.H.B.; Xu, J. Secure Consistency Verification for Untrusted Cloud Storage by Public Blockchains. In Proceedings of the Security and Privacy in Communication Networks—15th EAI International Conference, SecureComm 2019, Orlando, FL, USA, 23–25 October 2019; Volume 304; pp. 39–62.
15. Herlihy, M.; Wing, J.M. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* **1990**, *12*, 463–492. [CrossRef]
16. Lamport, L. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: The Works of Leslie Lamport*; Malkhi, D., Ed.; ACM: New York, NY, USA, 2019; pp. 179–196. [CrossRef]
17. Liskov, B.; Ladin, R. Highly Available Distributed Services and Fault-Tolerant Distributed Garbage Collection. In Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing, New York, NY, USA, 11–13 August 1986; pp. 29–39. [CrossRef]
18. Fidge, C.J. A Limitation of Vector Timestamps for Reconstructing Distributed Computations. *Inf. Process. Lett.* **1998**, *68*, 87–91. [CrossRef]
19. Mattern, F. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *J. Parallel Distrib. Comput.* **1993**, *18*, 423–434. [CrossRef]
20. DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P.; Vogels, W. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*; Association for Computing Machinery: New York, NY, USA, 2007; pp. 205–220. [CrossRef]
21. Lakshman, A.; Malik, P. Cassandra: A decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.* **2010**, *44*, 35–40. [CrossRef]
22. Ellis, J. Why Cassandra Doesn't Need Vector Clocks. 2013. Available online: https://www.datastax.com/blog/why-cassandra-doesnt-need-vector-clocks (accessed on 27 September 2021).
23. Corbett, J.C.; Dean, J.; Epstein, M.; Fikes, A.; Frost, C.; Furman, J.J.; Ghemawat, S.; Gubarev, A.; Heiser, C.; Hochschild, P.; et al. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* **2013**, *31*, 1–22.
24. Chang, F.; Dean, J.; Ghemawat, S.; Hsieh, W.C.; Wallach, D.A.; Burrows, M.; Chandra, T.; Fikes, A.; Gruber, R. Bigtable: A Distributed Storage System for Structured Data (Awarded Best Paper!). In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06), Seattle, WA, USA, 6–8 November 2006; pp. 205–218.
25. Kartch, R. Best Practices for Network Border Protection. 2017. Available online: https://insights.sei.cmu.edu/sei_blog/2017/05/best-practices-for-network-border-protection.html (accessed on 18 March 2021).
26. Kim, B.H.; Oh, S.; Lie, D. Consistency Oracles: Towards an Interactive and Flexible Consistency Model Specification. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*; Association for Computing Machinery: New York, NY, USA, 2017; pp. 82–87. [CrossRef]
27. Cooper, B.F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; Sears, R. Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, IN, USA, 10–11 June 2010; Hellerstein, J.M., Chaudhuri, S., Rosenblum, M., Eds.; ACM: New York, NY, USA, 2010; pp. 143–154. [CrossRef]
28. Cachin, C. Integrity and Consistency for Untrusted Services - (Extended Abstract). In Proceedings of the SOFSEM 2011: Theory and Practice of Computer Science—37th Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, 22–28 January 2011; Cerná, I., Gyimóthy, T., Hromkovic, J., Jeffery, K.G., Královic, R., Vukolic, M., Wolf, S., Eds.; Springer: New York, NY, USA, 2011, Volume 6543; pp. 1–14. [CrossRef]

29. Cachin, C.; Ohrimenko, O. Verifying the Consistency of Remote Untrusted Services with Commutative Operations. In Proceedings of the Principles of Distributed Systems—18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, 16–19 December 2014; Aguilera, M.K., Querzoni, L., Shapiro, M., Eds.; Springer: New York, NY, USA, 2014; Volume 8878, pp. 1–16. [CrossRef]
30. Maniatis, P.; Baker, M. Secure History Preservation Through Timeline Entanglement. In Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, 5–9 August 2002; pp. 297–312.
31. Haeberlen, A.; Aditya, P.; Rodrigues, R.; Druschel, P. Accountable Virtual Machines. In Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, Vancouver, BC, Canada, 4–6 October 2010; Arpaci-Dusseau, R.H., Chen, B., Eds.; USENIX Association: Berkeley, CA, USA, 2010; pp. 119–134.
32. Haeberlen, A.; Kouznetsov, P.; Druschel, P. PeerReview: Practical accountability for distributed systems. In Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, WA, USA, 14–17 October 2007; Bressoud, T.C., Kaashoek, M.F., Eds.; ACM: New York, NY, USA, 2007; pp. 175–188. [CrossRef]
33. Adya, A.; Bolosky, W.J.; Castro, M.; Cermak, G.; Chaiken, R.; Douceur, J.R.; Howell, J.; Lorch, J.R.; Theimer, M.; Wattenhofer, R. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, MA, USA, 9–11 December 2002; Culler, D.E., Druschel, P., Eds.; USENIX Association: Berkeley, CA, USA, 2002.
34. Chun, B.; Maniatis, P.; Shenker, S.; Kubiatowicz, J. Attested append-only memory: Making adversaries stick to their word. In Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, WA, USA, 14–17 October 2007; Bressoud, T.C., Kaashoek, M.F., Eds.; ACM: New York, NY, USA, 2007; pp. 189–204. [CrossRef]
35. Levin, D.; Douceur, J.R.; Lorch, J.R.; Moscibroda, T. TrInc: Small Trusted Hardware for Large Distributed Systems. In Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, Boston, MA, USA, 22–24 April 2009; Rexford, J., Sirer, E.G., Eds.; USENIX Association: Berkeley, CA, USA, 2009; pp. 1–14.
36. Schiffman, J.; Sun, Y.; Vijayakumar, H.; Jaeger, T. Cloud Verifier: Verifiable Auditing Service for IaaS Clouds. In Proceedings of the IEEE Ninth World Congress on Services, SERVICES 2013, Santa Clara, CA, USA, 28 June–3 July 2013; IEEE Computer Society: Los Alamitos, CA, USA, 2013; pp. 239–246. [CrossRef]
37. Stefanov, E.; van Dijk, M.; Juels, A.; Oprea, A. Iris: A scalable cloud file system with efficient integrity checks. In Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3–7 December 2012; Zakon, R.H., Ed.; ACM: New York, NY, USA, 2012; pp. 229–238. [CrossRef]
38. Goh, E.; Shacham, H.; Modadugu, N.; Boneh, D. SiRiUS: Securing Remote Untrusted Storage. In Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, CA, USA, 24–27 February 2003; The Internet Society: Reston, VA, USA, 2003.
39. Rivest, R.L.; Fu, K.; Fu, K.E. Group Sharing and Random Access in Cryptographic Storage File Systems. Master's Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1999.
40. Goodrich, M.T.; Papamanthou, C.; Tamassia, R.; Triandopoulos, N. Athos: Efficient Authentication of Outsourced File Systems. In Proceedings of the Information Security, 11th International Conference, ISC 2008, Taipei, Taiwan, 15–18 September 2008; Wu, T., Lei, C., Rijmen, V., Lee, D., Eds.; Springer: New York, NY, USA, 2008; Volume 5222, *Lecture Notes in Computer Science*; pp. 80–96. [CrossRef]
41. Barsoum, A.F.; Hasan, M.A. Enabling Dynamic Data and Indirect Mutual Trust for Cloud Computing Storage Systems. *IEEE Trans. Parallel Distrib. Syst.* **2013**, *24*, 2375–2385. [CrossRef]
42. Dobre, D.; Karame, G.; Li, W.; Majuntke, M.; Suri, N.; Vukolic, M. PoWerStore: Proofs of writing for efficient and robust storage. In Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, 4–8 November 2013; Sadeghi, A., Gligor, V.D., Yung, M., Eds.; ACM: New York, NY, USA, 2013; pp. 285–298. [CrossRef]
43. Vrable, M.; Savage, S.; Voelker, G.M. BlueSky: A cloud-backed file system for the enterprise. In Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, 14–17 February 2012; Bolosky, W.J., Flinn, J., Eds.; USENIX Association: Berkeley, CA, USA, 2012; p. 19.
44. Bessani, A.N.; Correia, M.P.; Quaresma, B.; André, F.; Sousa, P. DepSky: Dependable and secure storage in a cloud-of-clouds. European Conference on Computer Systems. In Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria, 10–13 April 2011; Kirsch, C.M., Heiser, G., Eds.; ACM: New York, NY, USA, 2011; pp. 31–46. [CrossRef]
45. Bowers, K.D.; Juels, A.; Oprea, A. HAIL: a high-availability and integrity layer for cloud storage. In Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, IL, USA, 9–13 November 2009; Al-Shaer, E., Jha, S., Keromytis, A.D., Eds.; ACM: New York, NY, USA, 2009; pp. 187–198. [CrossRef]
46. Juels, A.; Kaliski, B.S., Jr. Pors: Proofs of retrievability for large files. In Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, VA, USA, 28–31 October 2007; Ning, P., di Vimercati, S.D.C., Syverson, P.F., Eds.; ACM: New York, NY, USA, 2007; pp. 584–597. [CrossRef]
47. Chen, H.C.H.; Lee, P.P.C. Enabling Data Integrity Protection in Regenerating-Coding-Based Cloud Storage: Theory and Implementation. *IEEE Trans. Parallel Distrib. Syst.* **2014**, *25*, 407–416. [CrossRef]

48. Ateniese, G.; Burns, R.C.; Curtmola, R.; Herring, J.; Kissner, L.; Peterson, Z.N.J.; Song, D.X. Provable data possession at untrusted stores. In Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, VA, USA, 28–31 October 2007; Ning, P., di Vimercati, S.D.C., Syverson, P.F., Eds.; ACM: New York, NY, USA, 2007; pp. 598–609. [CrossRef]
49. van Dijk, M.; Juels, A.; Oprea, A.; Rivest, R.L.; Stefanov, E.; Triandopoulos, N. Hourglass schemes: How to prove that cloud files are encrypted. In Proceedings of the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, 16–18 October 2012; Yu, T., Danezis, G., Gligor, V.D., Eds.; ACM: New York, NY, USA, 2012; pp. 265–280. [CrossRef]