Don’t Trust the Cloud, Verify: Integrity and Consistency for Cloud Object Stores

MARCUS BRANDENBURGER, CHRISTIAN CACHIN and NIKOLA KNEŽEVIĆ, IBM Research - Zurich

Cloud services have turned remote computation into a commodity and enable convenient online collaboration. However, they require that clients fully trust the service provider in terms of confidentiality, integrity, and availability. Towards reducing this dependency, this paper introduces a protocol for verification of integrity and consistency for cloud object storage (VICOS), which enables a group of mutually trusting clients to detect data-integrity and consistency violations for a cloud object-storage service. It aims at services where multiple clients cooperate on data stored remotely on a potentially misbehaving service. VICOS enforces the consistency notion of fork-linearizability, supports wait-free client semantics for most operations, and reduces the computation and communication overhead compared to previous protocols. VICOS is based on a generic authenticated data structure. Moreover, its operations cover the hierarchical name space of a cloud object store, supporting a real-world interface and not only a simplistic abstraction. A prototype of VICOS that works with the key-value store interface of commodity cloud storage services has been implemented, and an evaluation demonstrates its advantage compared to existing systems.

Categories and Subject Descriptors: C.2.4 [Distributed systems]: Client/server, Distributed applications; D.4.6 [Security and Protection]: Cryptographic controls; H.3.4 [Systems and Software]: Distributed systems

General Terms: Algorithms, Security, Verification

Additional Key Words and Phrases: Cloud storage, data integrity, fork-linearizability, wait-freedom

1. INTRODUCTION

More and more data is outsourced to the cloud, and collaborating on a shared resource using cloud services has become easier than ever. Programmers work together on online source code repositories, global project teams produce technical deliverables, and friends share their photo albums. Nevertheless, the clients need to trust the cloud provider as they rely on it for the confidentiality and correctness of their data. Encryption may preserve the confidentiality of data but cannot prevent inadvertent or malicious data modifications. This work shows how to protect the integrity and consistency of data on an untrusted cloud storage service accessed by multiple clients.

With a single client only, the client may locally keep a short cryptographic hash value of the outsourced data. Later, this can be used to verify the integrity of the data returned by the cloud storage service. However, with multiple disconnected clients, no common synchronization, and no communication among the clients, neither hashing
nor digital signatures are sufficient by themselves. The reason is that a malicious or Byzantine server may violate the consistency of the data, for example, by reordering or omitting properly authenticated operations, so that the views of the storage state at different clients diverge. A malicious cloud server may pretend to one set of clients that some operations by others simply did not occur. In other words, freshness can be violated and the clients cannot detect such replay attacks until they communicate directly. The problem is particularly relevant in cryptographic online voting and for web certificate transparency [Laurie 2014].

The strongest achievable notion of consistency in this multi-client model is captured by fork-linearizability, introduced by Mazières and Shasha [2002]. A consistency and integrity verification protocol may guarantee this notion by adding condensed data about the causal evolution of the client’s views into their interaction with the server. This ensures that if the server creates only a single discrepancy between the views of two clients, these clients may never observe operations of each other afterwards. In other words, if the server ever lies to some clients and these clients communicate later, they will immediately discover the violation. Hence, with only one check they can verify a large number of past operations.

The SUNDR system [Li et al. 2004] pioneered fork-linearizable consistency and demonstrated a network file system protected by a hash tree of every user’s files. SUNDR uses an expensive protocol, requiring messages of size \( \Omega(n^2) \) for \( n \) clients [Cachin et al. 2007]. Like other systems providing fork-linearizability, it suffers from the inherent limitation that sometimes, even with a correct server, clients have to block and cannot proceed with their next operation because other clients are concurrently executing operations [Cachin et al. 2007].

In order to prevent blocking, FAUST [Cachin et al. 2011b], SPORC [Feldman et al. 2010], and Venus [Shraer et al. 2010] relax their guarantees to weak fork-linearizability, which establishes consistency only eventually, after further operations occur; this is not desirable because a client may only know later that a protocol output was not correct. SPORC and the related Blind Stone Tablet (BST) [Williams et al. 2009] protocol shift the maintenance of state to the clients, such that the server is merely responsible for coordination; every client holds a complete copy of the system’s state. This contradicts the goal of outsourcing data to the cloud. Another way to avoid blocking is explored by BST and COP [Cachin and Ohrimenko 2014]: they let any commuting operations proceed immediately.

In this paper, we present VICOS, a verification protocol for the integrity and consistency of cloud object storage, which overcomes these limitations and demonstrates a complete practical system. VICOS supports the optimal consistency notion of fork-linearizability, provides wait-free semantics for all compatible client operations, and has smaller overhead than previous protocols. The notion of compatible operations, as introduced here, generalizes the progress condition over commuting operations, which are considered in past work. Informally, an operation \( o \) is compatible with \( \omega \) if executing \( o \) before \( \omega \) does not influence the behavior of \( \omega \) (that is, the observed behavior of \( \omega \) and all future operations remains after \( \omega \) the same as if \( o \) was absent). Conceptually, VICOS is based on abstract authenticated data structures in a modular way. Moreover, it unifies the two different lines of work on this problem so far, namely, the untrusted storage protocols [Li et al. 2004; Cachin et al. 2007; Cachin et al. 2011b] that feature remote state and are based on vector clocks, and the remote service verification protocols [Williams et al. 2009; Feldman et al. 2010; Cachin and Ohrimenko 2014], which create local copies of the state and use hash chains. In particular, VICOS maintains state remotely, at the server, but uses hash chains for consistency verification. Furthermore, VICOS incurs only a constant communication overhead per operation,
independent of the number of clients, whereas FAUST and Venus require vector clocks of size $\Omega(n)$ with $n$ clients.

Similar to all previous work mentioned before, VICOS targets a group of mutually trusting clients that access an untrusted remote storage service. Malicious clients colluding with the provider could break its guarantees. As it seems unrealistic to extend mutual trust to all clients of a general-purpose commercial cloud storage platform, VICOS aims at protecting relatively small workgroups, where members trust each other, and ensures consistency among these clients.

We have implemented and evaluated VICOS with a commodity cloud object store; the results demonstrate that the overhead for distributed, multi-client integrity and consistency verification is acceptable for the strong consistency level that it provides. VICOS protects a complete cloud storage service (assuming it has atomic semantics), spanning many objects and offering read, write, delete, and directory listing operations; this stands in contrast to Venus [Shraer et al. 2010], which only provided consistency for a single data object. Furthermore, VICOS notifies the clients whenever the integrity or consistency is violated but does not address recovery operations.

1.1. Contributions
This work makes the following contributions towards ensuring integrity for data stored on untrusted cloud providers:

— A novel abstract protocol to verify the integrity and consistency of a generic service based on an authenticated data structure [Tamassia 2003], which ensures fork-linearizability, supports wait-free semantics for compatible operations, and incurs only constant communication overhead. This protocol also generalizes authenticated data structures to multiple writers.

— An instantiation of this protocol for a commodity cloud object store, called VICOS. It represents the first integrity protection protocol with all of the above features for the standard operations of a cloud object store.

— An implementation and evaluation of VICOS using COSBench, demonstrating its practicality. In particular, the overhead of VICOS for integrity protection remains acceptable small with moderate concurrency and increases slightly when many clients access the same data concurrently. The prototype is available open-source.

Let us also mention what VICOS does not aim at. As a consequence of its trust model, VICOS cannot protect the consistency of all data in a public, general-purpose cloud-storage service accessed by clients that have no relation to each other. Instead it offers integrity and consistency for a defined group of clients. Furthermore, many cloud storage systems employing wide-area replication offer high availability and eventual consistency, but not linearizability, due to the inherent limitations of geo-replicated systems as formulated in Brewer’s “CAP theorem” [Gilbert and Lynch 2002]. VICOS imposes strong consistency (linearizability and fork-linearizability, respectively) as a defense against attacks for services that usually appear consistent.

1.2. Organization
The paper continues with introducing the model and defines our notion of an authenticated data structure (ADS). Sec. 3 presents the abstract integrity protocol for ADS and discusses its properties. VICOS is introduced in Sec. 4 and the evaluation appears in Sec. 6. Related work is discussed in Sec. 7.

2. MODEL
We consider an asynchronous distributed system with $n$ mutually trusting clients $C_1, \ldots, C_n$ and a server $S$. The communication between the server and the clients
is reliable and respects first-in first-out (FIFO) semantics. Clients cannot communicate with each other. A protocol $P$ specifies the behavior of the clients and the server. All clients are correct and hence follow $P$; in particular, they do not crash (although crashes could be tolerated with extra measures in the protocols). The server is either correct and follows $P$ or Byzantine, deviating arbitrarily from $P$.

The clients invoke operations of a stateful functionality $F$, implementing a set of deterministic operations; $F$ defines a response and a state change for every operation. We use the standard notions of executions, histories, sequential histories, real-time order, concurrency, and well-formed executions from the literature [Attiya and Welch 2004]. In particular, every operation in an execution is represented by an invocation event and a response event. We extend $F$ with a special return value ABORT that allows an operation to abort without taking effect, which may be used when concurrent operations would cause it to block [Majuntke et al. 2009; Cachin and Ohrimenko 2014].

2.1. Consistency properties

When the server $S$ is correct, the protocols should provide the standard notion of linearizability [Herlihy and Wing 1990] with respect to $F$. It requires that an execution of operations by all clients together is equivalent to an imaginary sequential execution of $F$. More precisely, for an execution to be linearizable, all invocation and response events occurring at all clients can be permuted into one totally ordered sequence, where (1) every operation invocation is followed immediately by the corresponding response; (2) all operations are correct according to the specification of $F$; and (3) the operations in the sequence respect the real-time order among operations observed in the execution.

Fork-linearizability [Mazières and Shasha 2002; Cachin et al. 2007] relaxes this common global view and permits that the clients observe an execution that may split into multiple linearizable “forks,” which must never join again. More precisely, an execution is fork-linearizable when every client observes a linearizable history (containing all operations that the client executes itself) and for any operation observed by multiple clients, the history of events occurring before the operation is the same at those clients. This implies that if the views of the execution at two clients ever diverge, they cannot observe each other’s operations any more, and makes it easy to spot consistency violations by the server.

Furthermore, we recall the concept of a fork-linearizable Byzantine emulation [Cachin et al. 2007]. It requires that our protocol among the clients and the Byzantine server satisfies two conditions: (1) When the server is correct, then the service is linearizable; otherwise, (2) it is still fork-linearizable. Finally, our protocol may only abort (by returning ABORT) if there is some reason for this; in other words, when the clients execute operations sequentially, then no client ever aborts.

2.2. Cryptographic primitives

Our protocols use a cryptographic hash function and digital signature schemes for protecting data against modification. We model them in an idealized way, as if implemented by a distributed oracle [Cachin et al. 2011a].

A cryptographic hash function $hash$ maps a bit string $x$ of arbitrary length to a short, unique hash value $h$. Its implementation is deterministic and maintains a list $L$ of all $x$ that have been queried so far. When the invocation contains $x \in L$, then $hash$ responds with the index of $x$ in $L$; otherwise, $hash$ appends $x$ to $L$ and returns its index. This ideal implementation models only collision resistance, i.e., that it is not feasible to find two different values $x_1$ and $x_2$ such that $hash(x_1) = hash(x_2)$.

A digital signature scheme as used here provides two functions, $sign$, and $verify$, to ensure the authenticity of a message created by a known client. The scheme works
as follows: A client $C_i$ invokes $\text{sign}_i(m)$ with a message $m$ as argument and obtains a signature $\phi \in \{0, 1\}^*$ with the response. Only client $C_i$ can invoke $\text{sign}_i$. When $m$ and $\phi$ are sent to another client, that client may verify the integrity of $m$. It invokes $\text{verify}_i(\phi, m)$ and obtains $\text{TRUE}$ as a response if and only if $C_i$ has executed $\text{sign}_i(m)$; otherwise, $\text{verify}_i(\phi, m)$ returns FALSE. Every client, as well as $S$, may invoke verify.

2.3. Authenticated data structures

This section defines the model of authenticated data structures (ADS) used here. Authenticated data structures [Naor and Nissim 2000; Martel et al. 2004; Tamassia 2003] are a well-known tool for verifying operations and their results over data outsourced to untrusted servers. Popular instantiations rely on Merkle hash trees or other hierarchical authenticated search structures [Goodrich et al. 2011; Miller et al. 2014].

We model an ADS for an arbitrary deterministic functionality $F$. Departing from the literature on ADSs, we eliminate the special role of the single writer or “source” and let any client perform update operations; likewise, we unify queries and updates into one type of operation from a set $O$. Operations may contain arguments according to $F$, but these are subsumed into the different $o \in O$. The functionality specifies a state $s \in S$, which will be maintained by the server $S$, starting with an initial state $s_0$. For example, this may include all data stored on a cloud storage service. Given $s$, applying an operation $o$ of $F$ means to compute $(s', r) \leftarrow F(s, o)$, resulting in a new state $s' \in S$ and a response $r \in R$.

Operations are triggered by the clients and executed with the help of $S$. In order to verify the responses of $S$, a client stores an authenticator $a$, a short value also called a digest. Initially, the authenticator is a special value $a_0$.

For executing an operation $o$, the client asks $S$ to run an algorithm $\text{query}_F$,

$$(r, \sigma_o) \leftarrow \text{query}_F(s, o),$$

producing a response $r$ and auxiliary data $\sigma_o$ for $o$; the latter may serve as a proof for the validity of the response. The client obtains $r$ and $\sigma_o$ from $S$ and locally performs an operation $\text{authexec}_F$ to validate the response on the basis of the authenticator. From this the client obtains an output

$$(a', \sigma'_o, v) \leftarrow \text{authexec}_F(o, a, r, \sigma_o).$$

Here $a'$ and $\sigma'_o$ are the updated authenticator and auxiliary data, respectively, and $v \in \{\text{FALSE}, \text{TRUE}\}$ denotes a Boolean verification value that tells the client whether the response $r$ from $S$ is valid.

The client should then send $\sigma'_o$ back to $S$, so that the server may actually execute $o$ and update its state from $s$ to $s'$ by running $\text{refresh}_F$ as

$$s' \leftarrow \text{refresh}_F(s, o, \sigma'_o).$$

Note we may also consider these operations for sequences of operations.

An ADS [Naor and Nissim 2000; Martel et al. 2004; Tamassia 2003] is a special case of this formalization, in which the operations $O$ can be partitioned into update-operations $U$ and query-operations $Q$. Update operations generate no response, i.e., $F(s, u) = (s', \bot)$ for all $u \in U$ and queries do not change the state, that is, $F(s, q) = (s, r)$ for all $q \in Q$.

Furthermore, an ADS may contain initialization and key-generation routines and all algorithms may take public and private keys as inputs in addition. For simplicity, and because our ADS implementations are unkeyed, we omit them here.

An ADS satisfies correctness and security. Consider the execution of a sequence $(o_1, \ldots, o_m)$ of operations, $F(s_0, (o_1, \ldots, o_m))$, which means to compute $(s_j, r_j) \leftarrow$...
\( F(s_{j-1}, o_j) \) for \( j = 1, \ldots, m \). A **proper authenticated execution** of \( (o_1, \ldots, o_m) \) computes the steps

\[
(r_j, \sigma_{o,j}) \leftarrow \text{query}_F(s_{j-1}, o_j) \\
(a_j, \sigma'_{o,j}, v_j) \leftarrow \text{authexec}_F(o_j, r_{j-1}, r_j, \sigma_{o,j}) \\
s_j \leftarrow \text{refresh}_F(s_{j-1}, o_j, \sigma'_{o,j}).
\]

such that \( v_j = \text{TRUE} \), for \( j = 1, \ldots, m \).

An ADS is **correct** if the proper authenticated execution of any operation sequence \( (o_1, \ldots, o_m) \) outputs state \( s_m \) and response \( r_m \) such that \( (s_m, r_m) = F(s_0, (o_1, \ldots, o_m)) \).

Furthermore, an ADS must be **secure** against an adversary \( A \) that tries to forge a response and auxiliary data that are considered valid by a client. More precisely, \( A \) adaptively determines an operation sequence \( (o_1, \ldots, o_m) \), which is taken through a proper authenticated execution by a challenger; at every step, \( A \) obtains \( a_j \) and \( \sigma'_{o,j} \), and then determines \( o_{j+1} \) and whether the execution continues. Finally, after obtaining \( a_m \) and \( s_m \), \( A \) outputs an operation \( o^* \), a response \( r^* \), and a value \( \sigma'_o \). The ADS is secure if no \( A \) succeeds in creating \( o^*, r^*, s^*_m \), and \( \sigma'_o \) such that

\[
(\cdot, \cdot, \text{TRUE}) = \text{authexec}_F(o^*, a_m, r^*, \sigma'_o)
\]

but \( F(s_m, o^*) = (s_{m+1}, r_{m+1}) \) with \( r_{m+1} \neq r^* \). (The “don’t-care” symbol \( \cdot \) in the tuple indicates that only a subset of the tuple elements are needed.) In other words, \( A \) cannot find any \( o^* \) executed on \( s_n \), and forge a response \( r^* \) and a proof for \( o^* \) that is accepted by the client, unless the response is correct according to \( F \) and \( r^* = r_{m+1} \).

Note that this formalization represents an “idealized” security notion. It is easy to formulate an equivalent computational security condition using the language of modern cryptology [Goldreich 2001]. This model subsumes the one of Cachin [2011] and generalizes ADS [Tamassia 2003] to multiple writing clients, where here the authenticator is implicitly synchronized among the clients.

### 2.4. Compatible operations

Our protocol takes advantage of **compatible** operations that permit “concurrent” execution without compromising the goal of ensuring fork-linearizability to the clients. We say an operation \( o' \) is compatible with another operation \( o \) (in a state \( s \)) if the presence of \( o' \) before \( o \) does not influence the return value of \( o \) (in \( s \)). Compatible operations can be executed without blocking; this improves the throughput compared to earlier protocols, in particular with respect to COP [Cachin and Ohrimenko 2014], which considered the stronger notion of commutative operations.

Formally, we say an operation sequence \( \omega \) is **compatible** with an operation \( o \) in a state \( s \) whenever the responses of \( o \) remain the same regardless of whether \( \omega \) executed before \( o \). Hence, with

\[
(s', r) \leftarrow F(s, \omega); \quad (s'', p) \leftarrow F(s', o); \quad \text{and} \quad (t', q) \leftarrow F(s, o)
\]

it holds \( p = q \). Moreover, we say that \( \omega \) is **compatible** with \( o \) if and only if \( \omega \) is compatible with \( o \) in all states \( s \in S \) of \( F \). In the database literature [Weikum and Vossen 2002], our notion of compatibility is known as avoiding “write-read conflicts” or the absence of “dirty reads.”

In algorithms we use a function \( \text{compatible}_F \), which takes \( \omega \) and \( o \) as inputs and returns \( \text{TRUE} \) if and only if \( \omega \) and \( o \) are compatible.

Note that compatible operations are not necessarily commutative, but commutative operations are always compatible. For instance, a query operation \( q \) is compatible with
an update $u$ in any state; but when $q$ returns data modified by $u$, then $q$ and $u$ do not commute.

3. THE ADS INTEGRITY PROTOCOL (AIP)

This section introduces the ADS integrity protocol (AIP), a generic protocol to verify the integrity and consistency for any authenticated data structure (ADS) operated by a remote untrusted server. AIP extends and improves upon the commutative-operation verification protocol (COP) and its authenticated variant (ACOP) of Cachin and Ohrimenko [2014]. Sec. 4 shows how to instantiate AIP with an authenticated dictionary for protecting cloud storage; the result forms the core of VICOS.

3.1. Overview

Protocol AIP adopts the structure of previous protocols for verifying the consistency of an operation sequence executed by an untrusted server [Mazières and Shasha 2002; Cachin and Ohrimenko 2014]. In the simplest form, each client would obtain the complete history of operations from the server $S$, verify everything, append its own operation, sign the history again, and send it to $S$. But because this is infeasible in practice, the history is represented compactly by a hash chain or through a vector clock. Furthermore, since $S$ should not block one client $C_i$ while another, potentially slow client $C_j$ executes and authenticates its operation, $S$ should respond to $C_i$ right away. This means $C_i$ can only verify the history speculatively since $C_j$ has not yet verified and signed it. For a detailed illustration of such a simplified protocol, see the “bare-bones protocol” of Mazières and Shasha [2002] or the “lock-step protocol” of Cachin et al. [2007].

Every client in VICOS builds a hash chain over the history of all operations in its view. It includes the hash chain output in signed messages, which makes it easy for other clients to detect violations of consistency by the server. The processing of one operation in AIP is structured into an active and a passive phase, as shown in Fig. 1. The active phase begins when the client invokes an operation and ends when the client completes it and outputs a response; this takes one message roundtrip between the client and the server. Different from past protocols, the client stays further involved with processing authentication data for this operation during the passive phase, which is decoupled from the execution of further operations.

More precisely, when client $C_i$ invokes an operation $o \in \mathcal{O}$, it sends a signed INVOKE message carrying $o$ to the server $S$. The server assigns a sequence number $(t)$ to $o$ and responds with a REPLY message containing a list of pending operations, the response...
computed according to functionality $F$, an authenticator, and auxiliary data needed by the client for verification. Operations are pending (for $o$) because they have been started by other clients and $S$ has ordered them before $o$, but $S$ has not yet finished processing them. We distinguish between pending-other operations, invoked by other clients, and pending-self operations, which $C_i$ has executed before $o$.

After receiving the REPL message, the client checks its content. If the pending-other operations are compatible with $o$, then $C_i$ verifies (informally speaking) the pending-self operations including $o$ with the help of the authenticator. The authenticator supposedly represents the current state held by $S$, it must be signed by the client whose operation produced the state and come with a hash chain output that matches the client's own hash chain. If these values are correct, $C_i$ proceeds and outputs the response immediately. Along the way $C_i$ verifies that all data received from $S$ satisfies the conditions to ensure fork-linearizability. An operation that terminates like this is called successful; alternatively, when the pending-other operations are not compatible with $o$, then $o$ aborts. In this case, $C_i$ returns the symbol ABORT. In any case, the client subsequently commits $o$ and sends a signed COMMIT message to $S$ (note that also aborted operations are committed in that sense). This step terminates the active phase of the operation. The client may now invoke the next operation or retry $o$ if it was aborted. Note that $C_i$ outputs the response of $o$: this must always be correct and consistent, i.e., respect fork-linearizability. The ordering, on the other hand, is speculative and assumes the pending operations will be executed and committed in this sequence.

Processing of $o$ continues with the passive phase. Its goal is to actually execute $o$ on the remote state and to authenticate it within the ordering on which $C_i$ speculated during the active phase. Hence, at some later time, as soon as the operation immediately preceding $o$ in the assigned order has terminated its own passive phase, $S$ sends an UPDATE-AUTH message with auxiliary data and the authenticator of the preceding operation to $C_i$. When $C_i$ receives this, it validates the message content, verifies the execution of $o$ except when $o$ aborts, and checks that the operations which were pending for $o$ have actually been executed and authenticated as claimed by $S$ in REPL. The client now computes and signs a new authenticator that it sends to $S$ in a COMMIT-AUTH message. We say that $C_i$ authenticates $o$ at this time. When $S$ receives this message, then it applies $o$ by executing it on the state and stores the corresponding authenticator; this completes the passive phase of $o$.

Note that the server may receive COMMIT messages in a different order than assigned by the global sequence numbers, due to asynchrony. Still, the authentication steps in the passive phases of the different operations must proceed according to the assigned operation order. For this reason, the server maintains a second sequence number ($b$), which indicates the last authenticated operation that the server has applied to its state. Hence, $S$ buffers the incoming COMMIT messages and runs the passive phases sequentially in the assigned order.

Every client needs to know about all operations that the server has executed for checking consistency, and in particular for verifying UPDATE-AUTH messages. Therefore, when $S$ responds to the invocation of an operation by $C_i$, it includes in the REPL message a summary ($b$) of all authenticated operations that $C_i$ has missed since it last executed an operation. Prior to committing $o$, the client verifies these signatures and thereby clears the operations. The client also extends its hash chain with these operations.

### 3.2. Notation

The protocol is shown in Alg. 1–3 and formulated reactively. The clients and the server are state machines whose actions are triggered by events such as receiving messages. An ordered list with elements $e_1, e_2, \ldots, e_k$ is denoted by $E = \langle e_1, e_2, \ldots, e_k \rangle$; the ele-
ment with index \( j \) may be accessed as \( E[j] \). We also use maps that operate as associative arrays and store values under unique keys. A value \( v \) is stored in a map \( H \) by assigning it to a key \( k \), denoted by \( H[k] \leftarrow v \); for non-assigned keys, the map returns \( \bot \). The symbol \( \| \) denotes the concatenation of bit strings. The \texttt{assert} statement, parameterized by a condition, catches an error and immediately terminates the protocol when the condition is false. Clients use this to signal that the server misbehaved.

3.3. Data structures

This section describes the data structures maintained by every client and by the server. For simplicity, the pseudo code does not describe garbage collection, but we note where this is possible.

Every client \( C_i \) (Alg. 1) stores the sequence number of its last cleared operation in a variable \( c \). The hash chain \( H \) represents the condensed view that \( C_i \) has of the sequence of all operations. It is computed over the sequence of all applied operations and the sequence of pending operations announced by \( S \). Formally, \( H \) is a map indexed by operation sequence number; an entry \( H[t] \) is equal to \( \text{hash}(H[t-1]|o||I||j) \) when the \( l \)-th operation \( o \) is executed by \( C_j \), with \( H[0] = \text{NULL} \). Variable \( Z \) is a map that represents the status (SUCCESS or ABORT) of every operation, according to the result of the test for compatibility. The client only needs the entries in \( H \) and \( Z \) with indices greater than \( c \) and may garbage-collect older entries. Finally, \( C_i \) uses a variable \( u \) that is set to \( o \) whenever \( C_i \) has invoked operation \( o \) but not yet completed it; otherwise \( u \) is \( \bot \).

The server (Alg. 3) maintains the sequence number of the most recently invoked operation in a counter \( t \) and that of the most recently applied operation in \( b \). Every invoked operation is stored in a map \( I \) and every committed operation in a map \( O \); both maps are indexed by sequence number. The server only needs the entries in \( I \) with sequence numbers greater than \( b \). An entry in \( O \), at a sequence number \( b \) or greater, has to be stored until every client has committed some operation with a higher sequence number and may be removed later. Most importantly, the server keeps the state \( s \) of the ADS for \( F \), which reflects all successful operations up to \( b \). The server also stores the signed authenticator for every operation in a map \( A \) indexed by sequence number.

3.4. The protocol in detail

This section describes the ADS integrity protocol (AIP) as shown in Alg. 1–3. AIP is parameterized by an ADS and a functionality \( F \) that specifies its operations through \texttt{query}_F, \texttt{authexec}_F, and \texttt{refresh}_F. The client invokes AIP with an ADS-operation \( o \) by calling \texttt{aip-invoke}(\( o \)); it completes when AIP executes \texttt{return} at the end of the handler for the \texttt{REPLY} message. This ends the active phase of AIP, and the passive phase continues asynchronously in the background.

3.4.1. Active phase. When client \( C_i \) invokes an operation \( o \) (Alg. 1, \texttt{function aip-invoke}), it computes an \texttt{INVOKE}-signature \( \tau \) over \( o \) and \( t \); this proves to other clients that \( C_i \) has invoked \( o \). Then \( C_i \) stores \( o \) in \( u \) and sends an \texttt{INVOKE} message with \( o \) and \( \tau \) to the server.

When receiving this (Alg. 3, \texttt{upon} receiving \texttt{INVOKE}), the server increments the sequence number \( t \), assigns it to \( o \), and assembles the \texttt{REPLY} message for \( C_i \). First, \( S \) includes in \( \delta \) the applied operations that \( C_i \) has missed for clearing. Then it stores \( o \) and \( \tau \) in \( I[t] \); the value \( t \) is also called the \texttt{position} of \( o \). The pending operations \( \omega \) for \( o \) are found in \( I[b+1], \ldots, I[t] \), starting with the oldest non-applied operation, and include \( o \). In order to compute the response and the auxiliary data for \( o \) from the correct state, the server must then extract the \texttt{successful} pending-self operations \( \mu \) of \( C_i \).

Function \texttt{separate-pending}, shown next, splits the pending operations into pending-self and pending-others. This method is common to the server and the clients.
The server finds the status of a pending-self operation \( o' \) of \( C_i \) in \( O[b+k] \) because \( C_i \) has already committed \( o' \) prior to invoking \( o \) and because the messages between \( C_i \) and \( S \) are FIFO-ordered. On the other hand, \( C_i \) retrieves the status of \( o' \) from \( Z[b+k] \).

Then \( S \) computes the response \( r \) and auxiliary data \( \sigma_o \) via \( \text{query}_F(s, \mu) \); the response takes into account the state reached after the successful pending-self operations \( (\mu) \) of \( C_i \) but excludes the pending-other operations \( \gamma \) present in \( \omega \). The client will only execute \( o \) and output \( r \) when \( \gamma \) is compatible with \( o \) and, therefore, \( C_i \) is guaranteed a view in which the operations of \( \gamma \) occur after \( o \). This will ensure fork-linearizability. The \( \text{REPLY} \) message to \( C_i \) also includes \( A[b] \), the authenticator and \( \text{AUTH} \)-signature of the operation at position \( b \). For the very first operation (at position \( b = 0 \)), the protocol uses the initial authenticator \( a_0 \) of the ADS. The client passes these values to \( \text{authexec}_F(\mu, \cdot) \) for verifying the correctness of the response. Furthermore, the \( \text{REPLY} \) message contains \( \delta \), the list of all operations that have been authenticated (each one by the client that invoked it) and applied by \( S \) since \( C_i \)'s last operation. In particular, when \( c \) is the sequence number from the \( \text{INVOKE} \) message, \( \delta \) contains the operations at sequence numbers \( c + 1, \ldots, b \), but \( \delta \) may also be empty.

After receiving the \( \text{REPLY} \) message from \( S \) (Alg. 1), the client (1) processes and clears the authenticated operations in \( \delta \), (2) verifies the pending operations in \( \omega \), and (3) verifies that \( r \) is the correct response for \( o \). These steps are explained next.

For step (1), \( C_i \) first calls function \( \text{clear-authenticated} \) (Alg. 2), which verifies and/or extends the hash chain for every operation in \( \delta \) and checks the corresponding \( \text{COMMIT} \)-signature. This ensures for any operation which has been pending for \( C_i \) that the \textit{same} operation was committed and authenticated by its originator at the \textit{same} position where \( C_i \) saw it. If successful, all operations in \( \delta \) are cleared and \( C_i \)'s operation counter \( c \) is advanced to the end of \( \delta \). After executing \( \text{clear-authenticated} \), \( C_i \) also checks the \( \text{AUTH} \)-signature \( \psi \) on the authenticator \( a \) (these are in \( \alpha \) from the \( \text{REPLY} \)).

The client continues with step (2) in \( \text{check-pending} \) (Alg. 2) by verifying that the pending operations are announced correctly: for every operation in \( \omega \), \( C_i \) verifies the \( \text{INVOKE} \)-signature \( \tau \) and checks it against the hash chain \( H \). If this validation succeeds, it means the operation is consistent with a pending operation sent previously by \( S \). After iterating through the pending operations, the client checks also that the last operation in \( \omega \) is indeed its own current operation \( o \).

Subsequently (3) \( C_i \) invokes \( \text{separate-pending} \) on \( \omega \) to extract \( \mu \) and \( \gamma \), the pending-self and pending-other operations (see earlier). Then, \( C_i \) checks whether \( \gamma \) is compatible with \( o \) (stored in \( w \)). If yes, \( C_i \) calls \( \text{authexec}_F(\mu, a, r, \sigma_o) \) for verifying that applying \( \mu \) yields response \( r \) (recall that \( \mu \) includes \( o \) at the end). The goal of this step is only to check the correctness of the response, as the other outputs of \( \text{authexec}_F \) are ignored.

Finally, \( C_i \) commits \( o \) by generating a \( \text{COMMIT} \)-signature \( \phi \) over \( t \), the sequence number of \( o \), its status, and its hash chain entry, sends a \( \text{COMMIT} \) message with all of this to \( S \), and outputs the response \( r \).
Algorithm 1 ADS integrity protocol (AIP) for client $C_i$

**state**

$c \in \mathbb{N}_0$: sequence number of last cleared operation, initially 0

$H : \mathbb{N}_0 \rightarrow \{0, 1\}^*$: the hash chain, initially only $H[0] = \text{NULL}$

$Z : \mathbb{N}_0 \rightarrow \mathbb{Z}$: status map, initially empty

$u \in O \cup \{\perp\}$: current operation or $\perp$ if none, initially $\perp$

**function** `aip-invoke(o)`

- `u \leftarrow o`  // client does not invoke further op. before obtaining $o$’s response

- `τ \leftarrow \text{sign}_\tau(\text{INVOLVE}[o][i])`

- send message $[\text{INVOLVE}, o, τ, c]$ to $S$

**upon** receiving message $[\text{REPLY}, δ, b, α, ω, r, σ_o]$ from $S$

**clear-authenticated(δ, b)**  // catch up on missed operations

- $(a, ψ, j) \leftarrow α$  // authenticator of last operation applied by $S$

**assert** $(c = 0 \land b = c \land a = a_o) \lor \text{verify}_j(ψ, \text{AUTH}||b||H[b][α])$  // verify last authenticator

**check-pending(ω)**  // verify pending operations

- $(μ, γ) \leftarrow \text{separate-pending}(i, ω)$  // $μ$: successful pending-self ops.; $γ$: pending-other ops.

- $t \leftarrow b + \text{length}(ω)$

**if** compatible$(γ, u)$ **then**

- $(v, Z) \leftarrow \text{authexec}_F(μ, a, r, σ_o)$  // verify the response computed by $S$

**assert** $v$

- $Z[i] \leftarrow \text{SUCCESS}$

**else**

- $Z[i] \leftarrow \text{ABORT}$

- $τ \leftarrow \text{ABORT}$

- $φ \leftarrow \text{sign}_φ(\text{COMMIT}||t||u||i||Z[t]||H[t])$

- send message $[\text{COMMIT}, u, t, Z[t], φ]$ to $S$

- $u \leftarrow \perp$

**return** $r$  // return response of client operation $o$, matching `aip-invoke(o)`

**upon** receiving message $[\text{UPDATE-AUTH}, o, r, σ_o, φ, q, α]$ from $S$

**assert** \text{verify}_i(φ, \text{COMMIT}||q||α||i||Z[q]||H[q])  // authenticator of last operation applied by $S$

**assert** $(q = 1 \land c = 0 \land a = a_0) \lor (q > 1 \land \text{verify}_j(ψ, \text{AUTH}||q - 1||H[q - 1][α]))$

**if** $Z[q] = \text{SUCCESS}$ **then**

- $(a', σ_o', v) \leftarrow \text{authexec}_F(o, a, r, σ_o)$

**assert** $v$

**else**

- $(a', σ_o') \leftarrow (a, \perp)$

- $ψ' \leftarrow \text{sign}_φ(\text{AUTH}||q||H[q][α'])$

- send message $[\text{COMMIT-AUTH}, a', σ_o', ψ']$ to $S$

3.4.2. Passive phase. During the passive phase, the server receives the COMMIT message and sends the UPDATE-AUTH message, the client authenticates the operation and sends COMMIT-AUTH, the server receives this and updates its state accordingly.

More precisely, the server (Alg. 3) stores the content of all incoming COMMIT messages in $O$ and processes them in the order of their sequence numbers, indicated by $b$. When an operation with sequence number $b+1$ has been committed but not yet authenticated by the client and thus not applied by $S$ (Alg. 3, **upon** $O[b+1] \neq \perp \land A[b+1] = \perp$), the server uses $\text{query}_F$ to compute the response $r$ and auxiliary data $σ_o$ from the current state $s$. It sends this in an UPDATE-AUTH message to $C_i$, also including the operation at position $b$ (from $O[b]$) and its signed authenticator (taken from $A[b]$). These
Algorithm 2 ADS integrity protocol (AIP) for client $C_i$, continued

function $extend-chain(o, l, j)$ \hspace{1em} // extend $H$ with $o$ by $C_j$ at position $l$
if $H[l] = \bot$
\hspace{2em} $H[l] \leftarrow hash(H[l-1]||o||l||j)$ \hspace{1em} // extend by one
else if $H[l] \neq hash(H[l-1]||o||l||j)$ then 
\hspace{2em} if op. already present, it must be the same
\hspace{4em} return FALSE \hspace{2em} // server replies are inconsistent
\hspace{2em} return TRUE

function $clear-authenticated(\delta, b)$ \hspace{1em} // verify authenticated ops. in $\delta$ and clear them
assert $b = c + \text{length}(\delta)$
for $k = 1, \ldots, \text{length}(\delta)$ do \hspace{1em} // if $\delta$ is empty, skip the loop
\hspace{2em} $\langle o, z, \phi, j \rangle \leftarrow \delta[k]$
assert $extend-chain(o, c + k, j)$
assert $\text{verify}_j(\phi, \text{COMMIT}\|c + k\|o\|j\|z\|H[c + k])$
$c \leftarrow b$ \hspace{1em} // all operations in $\delta$ have been cleared

function $check-pending(\omega)$
assert $\text{length}(\omega) \geq 1$ \hspace{1em} // $\omega$ must contain at least the current operation ($u$)
for $k = 1, \ldots, \text{length}(\omega)$ do
\hspace{2em} $\langle o, \tau, j \rangle \leftarrow \omega[k]$
assert $extend-chain(o, c + k, j)$
assert $\text{verify}_j(\tau, \text{INVOKE}\|o\|j)$
assert $o = u \land j = i$ \hspace{1em} // variables $o$ and $j$ keep their values

values allow the client to verify the authenticity of the response for the operation at position $b + 1$.

When $C_i$ receives this UPDATE-AUTH message for $o$ and sequence number $q$ (Alg. 1), it first validates the message contents. In particular, $C_i$ verifies that the authenticator $a$ is covered by a valid AUTH-signature $\psi$ by client $C_j$ at position $q - 1$, using $C_j$’s hash chain entry $H[q - 1]$.

Next, if $o$ was not aborted, i.e., $Z[q] = \text{SUCCESS}$, the client invokes authexec$_F$ to verify that the auxiliary data and the response are correct, and to generate new auxiliary data $s'_o$ and a new authenticator $a'_o$, which vouches for the correctness of the state updates induced by $o$. Otherwise, $C_i$ skips this step, as the authenticator does not change. Then $C_i$ issues an AUTH-signature $\psi'_o$ and sends it back to $S$ together with $a'_o$ and $s'_o$ in a COMMIT-AUTH message.

As the last step in the passive phase, server $S$ (Alg. 3) receives COMMIT-AUTH, increments $b$, stores the signed authenticator in $A[b]$, and if the operation did not abort, $S$ applies it to $s$ through refresh$_S$. A future REPLY or UPDATE-AUTH message from $S$ will contain this signed authenticator from $A[b]$, to reflect the current state held by $S$.

3.5. Remarks

As in BST [Williams et al. 2009] and in COP [Cachin and Ohrimenko 2014], operations that do not interfere with each other may proceed without blocking. More precisely, if some pending operation is not compatible with the current operation, the latter is aborted and must be retried later. Preventing clients from blocking is highly desirable but cannot always be guaranteed without introducing aborts [Cachin et al. 2007]. The potential for blocking has led other systems, including SPORC [Feldman et al. 2010] and FAUST [Cachin et al. 2011b], to adopt weaker and less desirable guarantees than fork-linearizability.

Obviously, it makes no sense for a client to retry its operation while the non-compatible operation is still pending. However, the client does not know when the
Algorithm 3 ADS integrity protocol (AIP) for server $S$

**state**
- $t \in \mathbb{N}_0$: sequence number of last invoked operation, initially 0
- $b \in \mathbb{N}_0$: sequence number of last applied operation, initially 0
- $I : \mathbb{N} \rightarrow O \times \{0,1\}^* \times \mathbb{N}$: invoked operations, initially empty
- $O : \mathbb{N} \rightarrow O \times Z \times \{0,1\}^* \times \mathbb{N}$: committed operations, initially empty
- $A : \mathbb{N}_0 \rightarrow \{0,1\}^* \times \{0,1\}^* \times \mathbb{N}$: signed authenticators, initially $A[0] = (a_0, \bot, \bot)$
- $s \in \{0,1\}^*$: state of the service, initially $s = s_0$

**upon receiving message** [INVOKE, $o, \tau, c$] from $C_i$ **do**
- $t \leftarrow t + 1$
- $I[t] \leftarrow (o, \tau, i)$
- $\delta \leftarrow (O[c+1], \ldots, O[b])$ // operations missed by $C_i$; if $c = b$, then $\delta$ is empty
- $\omega \leftarrow (I[b+1], I[b+2], \ldots, I[t])$ // all pending operations
- $(\mu, \cdot) \leftarrow \text{separate-pending}(i, \omega)$
- $(r, \sigma_o) \leftarrow \text{query}_P(s, \mu)$ // compute response $r$ and auxiliary data $\sigma_o$ for $o$
- send message [REPLY, $\delta, b, A[b], \omega, r, \sigma_o$] to $C_i$

**upon receiving message** [COMMIT, $o, q, z, \phi$] from $C_i$ **do**
- $O[q] \leftarrow (o, z, \phi, i)$ // buffer it until $o$ comes up for authentication

**upon** $O[b+1] \not= \bot \land A[b+1] = \bot$ **do** // the next committed but not yet authenticated op.
- $(o, z, \cdot, j) \leftarrow O[b+1]$
- if $z = \text{SUCCESS}$ then
  - $(r, \sigma_o) \leftarrow \text{query}_P(s, o)$ // compute response and auxiliary data again
- else
  - $(r, \sigma_o) \leftarrow (\bot, \bot)$
- send message [UPDATE-AUTH, $o, r, \sigma_o, \phi, b+1, A[b]$] to $C_j$

**upon receiving message** [COMMIT-AUTH, $o, \sigma_o, \psi$] from $C_i$ **do**
- $b \leftarrow b + 1$
- $A[b] \leftarrow (a, \psi, i)$ // store authenticator $a$ and AUTH-signature $\psi$
- $(o, z, \cdot, j) \leftarrow O[b]$
- if $z = \text{SUCCESS}$ then
  - $s \leftarrow \text{refresh}_P(s, o, \sigma_o)$ // actually apply $o$ to the state

contending operation commits. Additional communication between the server and the clients could be introduced to signal this. Alternatively, the client may employ a probabilistic waiting strategy and retry after a random delay.

In the following we assume that $S$ is correct. The communication cost of AIP amounts to the five messages per operation. Every client eventually learns about all operations of all clients, as it must clear them and include them in its hash chain. However, this occurs only when the client executes an operation (in REPLY). At all other times between operations, the client may be offline and inactive. In a system with $n$ clients that performs $h$ operations in total, BST [Williams et al. 2009] and COP [Cachin and Ohrimenko 2014] require $\Theta(nh)$ messages overall. AIP reduces this cost to $\Theta(h)$ messages, which means that each client only processes a small constant number of messages per operation.

The size of the INVOKE, COMMIT, UPDATE-AUTH, and COMMIT-AUTH messages does not depend on the number of clients and on the number operations they execute. Note that in contrast to some earlier generic systems, such as BST [Williams et al. 2009], messages in VICOS do not include the complete state of the service. The size of the REPLY message is influenced by the amount of contention, as it contains the pending
operations. If one client is slow, the pending operations may grow with the number of further operations executed by other clients. Note that the oldest pending operation is the one at sequence number \( b + 1 \); hence, all operations ordered afterwards are treated as pending, even when they already have been committed. The \texttt{REPLY} message can easily be compressed to constant size, however, by omitting the pending operations that have already been sent in a previous message to the same client. See the protocol extensions in Sec. 3.7 for further discussion.

The functionality-dependent cost, in terms of communicated state and auxiliary data, is directly related to the ADS for \( F \). In practice, hierarchical authenticated search structures, such as hash trees and authenticated skip lists, permit small authenticators and auxiliary data [Crosby and Wallach 2011].

3.6. Correctness

We consider three cases: (1) \( S \) is correct and the clients execute operations (1a) sequentially or (1b) concurrently; and (2) \( S \) is malicious.

In case (1a), all operations execute one after each other. When, furthermore, the \texttt{COMMIT-AUTH} message from a client reaches \( S \) before the next operation is invoked, then AIP is similar to “serialized” SUNDR [Li et al. 2004] and the “lock-step protocol” of Cachin [2011]. This means that a client \( C_i \) executing an operation \( o \) receives a \texttt{REPLY} message with all authenticated operations that exist in the system and only \( o \) as pending operation. Then \( t = b + 1 \), and later \( C_i \) commits \( o \) and authenticates \( o \) without further operations intermixed at \( S \) nor at any client. Clearly, this execution is linearizable and satisfies the first condition of a fork-linearizable Byzantine emulation.

In case (1b), there may exist pending-other operations, but since \( C_i \) verifies that its own operations are compatible with them, the response value is correct. As \( S \) is correct, the views of all clients are equal, i.e., prefixes of each other, and this ensures linearizability.

For case (2), note that every client \( C_i \) starts to extend its view from a cleared, authenticated operation and the corresponding signed authenticator \( a \). If the pending-other operations in \( \gamma \) are compatible with \( o \) and the response is valid w.r.t. \( a \) (according to \texttt{authexec}_F), then it is safe for \( C_i \) to output the response and thereby include it into its view. The malicious \( S \) may order the operations in \( \gamma \) differently at other clients, creating a fork, but they can be omitted from the view of \( C_i \). The hash chain maintained by \( C_i \) contains a condensed representation of its entire view. By using its own hash chain entry during the verification of the \texttt{COMMIT} and \texttt{AUTH} signatures of other clients, \( C_i \) ensures that the views of these other clients are equal. Hence, whenever an operation \( o \) appears in the views of two clients, also their views are the same up to \( o \). This ensures fork-linearizability.

3.7. Extensions

In order to keep the complexity of the protocol description for AIP at a comprehensible level, we present important efficiency improvements informally here.

\textit{Compressing operation lists.} Several operation lists are sent multiple times, such as the pending operations in the \texttt{REPLY} message. An efficient implementation will only send the differences. Furthermore, one may remove aborted operations from being considered as pending to speed up the processing. Recall that an operation \( o \) remains pending until client authenticates it and server applies it, even if \( o \) was aborted. This has the drawback that later operations may not be compatible with \( o \) and abort unnecessarily. However, if \( o \) was aborted, the client has signed this, and \( S \) has received the \texttt{COMMIT} message, then \( S \) can include this with the list of pending operations of a later operation \( o' \). The client executing \( o' \) will take into account that \( o \) was aborted and
Integrity and Consistency for Cloud Object Stores

ignore it for determining whether \( \sigma' \) is compatible with the pending operations. This reduces the likelihood of further aborts.

**Batching and delegating operation authentication.** Recall that clients authenticate operations in the order of the server-assigned sequence numbers. A client \( C_{\text{slow}} \) may fall behind, and when faster clients execute more operations, the number of pending operations grows continuously. This creates much more work for the faster clients for processing the REPLY message and slows them down.

However, since all clients trust each other, another client \( C_{\text{fast}} \) may step in for \( C_{\text{slow}} \), handle the UPDATE-AUTH message, and sign the authenticator for the operation of \( C_{\text{slow}} \). Only small modifications to the data structures are needed to accommodate this change. Ideally \( C_{\text{fast}} \) has more processing power or is closer to \( S \) on the network than \( C_{\text{slow}} \); this choice should be determined heuristically based on actual performance.

Extending the above idea, \( S \) may actually batch all non-authenticated operations when an operation from \( C_i \) commits at position \( q \). Hence, \( S \) sends the UPDATE-AUTH messages for all operations between \( b \) and \( q \) to \( C_i \) and delegates the step of authenticating them to \( C_i \). This works because a client can authenticate two consecutive operations without going back to \( S \). The server then records the COMMIT-AUTH responses from the fastest client.

**Passive phase only for update operations.** Recall that the operations of \( F \) can be separated into query and update operations (\( Q \) and \( U \), respectively). Queries do not change the state; as is easy to see, they are compatible with every subsequent operation. But the passive phase of AIP is only needed for creating a new authenticator, after the state has changed. Therefore we can eliminate the passive phase for all queries; this considerably improves the efficiency of the protocol for read-intensive applications.

In particular, for every query \( o \in Q \), the passive phase is skipped and the verification operations are adjusted accordingly. When a client \( C_i \) has committed a query operation, the server immediately “applies” it and does not send an UPDATE-AUTH message later. For implementing this, the server has to maintain another variable \( d \) with the sequence number of the operation that most recently modified the state. The REPLY and UPDATE-AUTH messages now contain the operation \( O[d] \) that allows clients to verify the corresponding authenticator \( A[d] \).

**Tolerate client crashes.** In order to tolerate crashes in practice, we reuse the second extension. Since a client \( C_{\text{fail}} \) may crash before it finishes both phases, the server will never receive the missing message(s) and the operation is never authenticated. Suppose that another client \( C_i \) could detect the crash of \( C_{\text{fail}} \). Then \( C_i \) may take over for \( C_{\text{fail}} \), and commit and authenticate its operations. It is important that \( C_{\text{fail}} \) must not execute any operations again later, hence the failure detection should be reliable; if \( C_{\text{fail}} \) may join again later, it must be given a new identity. Group management protocols for adding and removing clients dynamically have been discussed in the context of existing systems, such as Venus [Shraer et al. 2010] and SPORC [Feldman et al. 2010].

### 4. Verification of Integrity and Consistency of Cloud Object Storage (VICOS)

We are now ready to introduce our main contribution, the protocol for verifying the integrity and consistency of cloud object storage, abbreviated VICOS. It leverages AIP from the previous section and provides a fork-linearizable Byzantine emulation for a practical object-store service, in a manner that is transparent to the storage provider.

We first define the operations of the cloud storage service and outline the architecture of VICOS. Next we instantiate AIP for verifying the integrity of a simple object store and show how VICOS extends this to practical cloud storage.
More precisely, VICOS consists of the following components (see Fig. 2):

1. A cloud object store (COS) with a key-value store interface, as offered by commercial providers. It maintains the object data stored by the clients using VICOS.

2. An AIP client and an AIP server, which implement the protocol from the previous section for the functionality of an authenticated dictionary (ADICT) and authenticate the objects at the cloud object store. The AIP server runs remotely as a cloud service accessed by the AIP client. This is abbreviated as AIP with ADICT.

3. The VICOS client exposes a cloud object store interface to the client application and transparently performs integrity and consistency verification. During each operation, the client consults the cloud object store for the object data itself and the AIP server for integrity-specific metadata. In particular, the ADICT in the AIP server stores the cryptographic hash of every object.

Note that the cloud object store as well as the AIP server are in the untrusted domain; they may, in fact, collude together against the clients.

4.1. Cloud object store (COS)
The cloud object store is modeled as a key-value store (KVS) and provides a “simple” storage service to multiple clients. It stores a practically unbounded number of objects in a flat namespace, where each object is an arbitrary sequence of bytes (or a “blob,” a binary large object), identified by a unique name or key. We assume that clients may only read and write entire objects, in contrast to file systems, for example.

Our formal notion of a KVS internally maintains a map \( M \) that stores the values in \( V \) under their respective keys taken from a universe \( K \). It provides four operations:

1. \( \text{kvs-put}(k, v) \): Stores a value \( v \in V \) under key \( k \in K \), that is, \( M[k] \leftarrow v \).
2. \( \text{kvs-get}(k) \): Returns the value stored under key \( k \in K \), that is, \( M[k] \).
3. \( \text{kvs-del}(k) \): Deletes the value stored under key \( k \in K \), that is, \( M[k] \leftarrow \bot \).
4. \( \text{kvs-list}() \): Returns a list of all keys for which a value is stored, that is, the list \( \{k \in K | M[k] \neq \bot\} \).

This API forms the core of many real-world cloud storage services, such as Amazon S3 or OpenStack Swift. Typically there is a bound on the length of the keys, such as a few hundred bytes, but the stored values can be much larger and practically unbounded (up to several Gigabytes). For simplicity, we assume that the cloud object store provides atomic semantics during concurrent access, being aware that cloud storage systems may only be eventually consistent [Bailis and Ghodsi 2013] due to network partitions. Its operations are denoted by \( \text{cos-put} \), \( \text{cos-get} \) etc.
Many practical cloud object stores support a single-level hierarchical name space, formed by containers or buckets. We abstract this separation into the keys here; however, a production-grade system would introduce this separation again by applying the design of VICOS for every container.

4.2. Authenticated dictionary (ADICT)

VICOS instantiates AIP with the functionality of a KVS that stores only short values. We refer to it as the authenticated dictionary, denoted by ADICT, with operations adict-put, adict-get, adict-del, and adict-list.

ADICT is a generic authenticated dictionary according to Naor and Nissim [2000] or Anagnostopoulos et al. [2001] (without support for multiple versions or “persistence,” however). It is implemented using the tree-based approach that descends from a Merkle tree, such as one of the schemes described by Crosby and Wallach [2011, Sec. 3]. The dictionary must support efficient ways to authenticate the absence of a key as well, for verifying operations that access non-existing keys. Such authenticated dictionaries support short “proofs” for all their operations, which are generally only logarithmic in the number of entries stored (except for an adict-list operation, obviously). More importantly, the query and refresh implementations for ADICT are of similar logarithmic complexity. The extra information needed for authentication is subsumed in the auxiliary data according to our ADS model of Sec. 2.3. The operations of ADICT follow the KVS definition above and are denoted by adict-put(k,v), adict-get(k), adict-del(k), and adict-list(). For the integration with AIP, the operation itself and its operands are parameters to queryADICT, authexecADICT, and refreshADICT.

The resulting authenticator is short, typically a single hash-function output.

A function compatibleADICT(µ,o) is needed for AIP to express the compatibility of the operations in the authenticated dictionary. The pairwise compatibility of ADICT, between a first (pending) operation and a second (current) operation is given by Table I. For instance, adict-put followed by adict-get for the same key or followed by adict-list are not compatible, whereas two adict-list and adict-get operations are always compatible. The function compatibleADICT(µ,o) extends this to a sequence µ and returns TRUE if and only if every operation in µ is compatible with o.

VICOS supports the same KVS interface and inherits this notion of compatibility for the cloud-storage operations. For more general services like databases, one would invoke a transaction manager here.

---

**Table I.** The compatibleADICT(·, ·) relation for ADICT and the KVS interface, where x, y ∈ K denote distinct keys, √ means compatible, and — means not compatible.
The advantage of considering operation compatibility over commutativity (as used in ACOP [Cachin and Ohrimenko 2014], for instance) becomes apparent here: only 8 pairs among the 49 cases shown are not compatible, whereas 22 among 49 cases do not commute and would be aborted with commutativity.

4.3. VICOS client implementation

VICOS emulates the key-value store API of a cloud object store (COS) to the client and transparently adds integrity and consistency verification. For clarity, its operations are denoted by vicos-put, vicos-get etc. As with AIP, consistency or data integrity violations committed by the server are detected through assert; any failing assertion triggers an alarm. It must be followed by a recovery action whose details go beyond the scope of this paper. Analogously to AIP, VICOS may return ABORT; this means that the operation was not executed and the client should retry it.

Algorithm 4 presents the pseudo code of the VICOS client. Basically, it protects every object in the COS by storing its cryptographic hash in the authenticated dictionary (ADICT). Operations trigger corresponding operations on COS and on ADICT, as provided by AIP for consistency enforcement.

Algorithm 4 Implementation of VICOS at the client.

function vicos-put($k, v$)
    $x \leftarrow$ a random nonce
    $cos$-put($k || x, v$)
    $h \leftarrow \text{hash}(v)$
    $r \leftarrow \text{aip-invoke(adict-put}(k, \langle x, h \rangle))$
    if $r = \text{ABORT}$ then // concurrent incompatible operation
        $cos$-del($k || x$)
    return $r$

function vicos-get($k$)
    $r \leftarrow \text{aip-invoke(adict-get}(k))$
    if $r = \text{ABORT}$ then // concurrent incompatible operation
        return $\text{ABORT}$
    $(x, h) \leftarrow r$
    $v \leftarrow \text{cos-get}(k || x)$
    assert $\text{hash}(v) = h$
    return $v$

function vicos-del($k$)
    $r \leftarrow \text{aip-invoke(adict-del}(k))$
    if $r = \text{ABORT}$ then // concurrent incompatible operation
        return $\text{ABORT}$
    $\text{cos-del}(k || *)$ // deletes all keys with prefix $k$
    return $r$

function vicos-list()
    $r \leftarrow \text{aip-invoke(adict-list}(())$
    if $r = \text{ABORT}$ then // concurrent incompatible operation
        return $\text{ABORT}$
    return $r$

In order to prevent race conditions, VICOS does not store an object under its key in COS directly, but translates every object key to a unique key for COS. Otherwise, two concurrent operations accessing the same object might interfere with each other and
leave the system in an inconsistent state. More precisely, in a \textit{vicos-put}(k, v) operation, the client chooses a nonce \(x\) (a value guaranteed to be unique in the system, such as a random string) and stores \(v\) in COS using \textit{cos-put}(k\|x, v), under the translated key \(k\|x\). Furthermore, it computes \(h ← \text{hash}(v)\) and stores \(⟨x, h⟩\) in ADICT under key \(k\). When \textit{adict-put} aborts due to concurrent operations, the client deletes \(v\) again from COS using \textit{cos-del}(k\|x).

For a \textit{vicos-get}(k) operation, the client first calls \textit{adict-get}(k) and retrieves \(⟨x, h⟩\). Unless this operation aborts, the client translates the key and calls \textit{cos-get}(k\|x) to retrieve the value \(v\). After \(v\) has been read, the client compares its hash value to \(h\), asserts that they match, and then outputs \(v\).

Without key translation, two concurrent \textit{vicos-put} operations \(o_1\) and \(o_2\) writing different values to the same key \(k\) might both succeed with \textit{cos-put}(k, \(v_1\)) and \textit{cos-put}(k, \(v_2\)), respectively, but the \textit{adict-put} for \(o_2\) might abort due to another concurrent operation. Then COS might store \(v_2\) but ADICT stores the hash of \(v_1\) and readers would observe a false integrity violation. Thanks to key translation, no versioning conflicts arise in the COS. Atomicity for multiple operations on the same object key follows from the properties of AIP with the ADICT implementation. The \textit{vicos-del}(k) and \textit{vicos-list}() operations proceed analogously; but the latter does not access COS.

4.4. Correctness

The implementation of VICOS satisfies the two properties of a fork-linearizable Byzantine emulation. First, when \(S\) is correct, then the clients proceed with their operations and all verification steps succeed. Hence, VICOS produces a linearizable execution. The linearization order is established by AIP running ADICT. Furthermore, when the clients execute sequentially, then by the corresponding property of AIP, no client ever receives \textsc{abort} from ADICT.

Second, consider the case of a malicious server controlling COS and the AIP server together. AIP ensures that the operations on ADICT (\textit{adict-put}, \textit{adict-get}, etc.) are fork-linearizable according to Sec. 4. The implementation of ADICT follows the known approach of memory checking with hash trees [Blum et al. 1994; Naor and Nissim 2000] and therefore authenticates the object hash values that VICOS writes to ADICT. According to the properties of the hash function, the object data is uniquely represented by its hash value. Since VICOS ensures the object data written to COS or returned to the client corresponds to the hash value stored in ADICT, it follows that all operations of VICOS are also fork-linearizable.

5. PROTOTYPE

We have implemented a prototype of VICOS in Java; it consists of a client-side library ("VICOS client") and the server code ("VICOS server"). The system can be integrated with applications that require cryptographic integrity and consistency guarantees for data in untrusted cloud storage services. It is available as open-source on GitHub (https://github.com/ibm-research/vicos).

5.1. VICOS implementation

The client-side library uses the BlobStore interface of Apache jclouds (Version 2.0.0 - Snapshot, https://jclouds.apache.org/) for connecting to different cloud object stores. The library supports streaming of large objects to and from COS. The server runs as a standalone web service, communicating with the client-side library using the Akka framework (Version 2.4.4, http://www.akka.io).

Akka is an event-driven framework which supports the actor model [Hewitt et al. 1973]. It fits perfectly into the system model of VICOS and allows for rapid develop-
ment. This simplifies the actual protocol implementation because of the high level of abstraction, especially relating to network operations and concurrency.

The client library as well as the server code are implemented as actors within the framework. Actors are independent units which can only communicate by exchanging messages. Every actor has a mailbox that buffers all incoming messages. By default messages are processed in FIFO order by the actor. This allows the server protocol implementation to process all incoming messages sequentially and execute each protocol step atomically, that is, mutually exclusive with respect to all others. The implementation of VICOS therefore closely follows the high-level description according to Sec. 3. Note that this clearly limits the systems performance by not utilizing modern multicore architecture, on the other hand.

We developed the VICOS client library so that it may easily be integrated into existing applications to provide integrity protection. It uses the modular approach of AIP instantiated with an ADS. A developer only needs to implement the desired ADS functionality, by defining the state, operations, and the compatibility relation of the ADS. For that reason, we have defined two interfaces: state and operation processor. Operations are described using Google’s Protocol Buffers (https://developers.google.com/protocol-buffers/) executed by the operation processor. This modular concept allows us to reuse and extend the core implementation of the protocol.

The VICOS prototype provides the ADICT functionality by implementing these interfaces. For simplicity of the implementation, the current prototype implements ADICT with a Merkle tree of depth 1. More precisely, the state is a map supporting get, put, del and list operations, and the authenticator is the hash computed over the map using a deterministic iterator. In a production-ready system, this will be replaced by one of the tree-based implementations of authenticated dictionaries mentioned in Sec. 4.2.

The ADICT operation processor provides the implementations of query, authexec, refresh, and compatible as described in Sec. 4.2. The client and the server protocol each contain an instance of the ADICT operation processor. The client exposes the KVS interface of the cloud object to the application, adding Java Exceptions for signaling integrity and consistency violations. Furthermore, the client library is completely asynchronous and supports processing the AIP passive phase in the background without blocking the client process.

The cryptographic signatures can be implemented in multiple ways. According to the security model, all clients trust each other, the server alone may act maliciously, and only clients issue digital signatures. Therefore, one can also realize AIP in a simpler way with “signatures” provided by a message-authentication code (MAC). For many applications, where strong mutual trust exists among the clients, MACs suffice and will result in faster execution. On the other hand, this optimization renders the system more fragile and exposes it more easily to attacks, should $S$ collude with a client against the other clients.

In particular, VICOS uses HMAC-SHA1 with 128-bit keys provided by the Java Cryptography Extension as the default signature implementation. The code also supports RSA and DSA signatures with 2048-bit keys. A user can choose between these implementations in a configuration file.

The core implementation of VICOS consists of $\sim$3400 sloc, the server part is $\sim$400 sloc more, whereas the client part including the integration with the evaluation platform (see below) takes $\sim$800 sloc extra.

5.2. Practical issues and optimizations

Bounded pending list. An issue that we discovered in Akka while implementing VICOS is Akka’s default maximum message size of only 128 kB. In particular, this
Integrity and Consistency for Cloud Object Stores

becomes a problem in VICOS when \textit{REPLY} messages include a large partial state or many pending operations, such as in runs with high contention. The direct limitation disappears when Akka is configured to allow larger message sizes. However, we experienced that large messages impact the overall performance negatively. When the number of pending operations increases, the resulting very large messages slow down the operations of VICOS.

Therefore, we implemented a way to bound the length of the pending-operations list, that is, we introduced a maximum number of pending operations as a configurable value and modified the protocol. When this maximum is reached, the server buffers all new incoming requests (\textit{INVOKE} messages) until enough other operations have completed and the number of pending operations goes below the limit. We tested with different maximum sizes for the pending list from 32 up to 1024 operations, and chose a limit of 128 for the evaluations. In particular, a \textit{REPLY} message for a \textit{put} operation containing the hash of a 10 kB data object and a single pending operation is about 380 B whereas the message size is about 19 kB with 128 pending operations.

A more robust solution for that issue would be to signal the clients to wait before sending more requests, instead of just buffering them at the server. Although limiting the number of pending operations under high server load increases the latency of client requests, it also increases the overall performance and stability of the system. In summary, we found that the benefits of this optimization outweigh its drawbacks.

\textit{Message delivery order.} During development we experienced a slowdown caused by the FIFO order in which the protocol actors process the arriving messages. Therefore, we implemented priority mailboxes for the server and the client actor and defined a priority rule to prefer \textit{COMMIT}, \textit{UPDATE-AUTH}, \textit{COMMIT-AUTH} messages over \textit{INVOKE} and \textit{REPLY} messages. This has the immediate benefit that the server processes \textit{UPDATE-AUTH} messages and thereby completes the \textit{PASSIVE PHASE} of already authenticated operations \textit{before} it starts working on new \textit{INVOKE} messages. This preference shortens the list of pending operations directly. Certainly, this may increase the response time for new operations again, but eventually it prevents more operations from aborting due to conflicts under high load.

6. EVALUATION

This section reports on performance measurements with the VICOS prototype. They study the general overhead of integrity protection, the scalability of the protocol, and the effect of (in-)compatible operations.

6.1. Experimental setup

The experiments use cloud servers and an OpenStack Swift-based object storage service (http://swift.openstack.org) of a major cloud provider (Softlayer – an IBM Company, http://www.softlayer.com/object-storage).

The VICOS server runs on a dedicated “baremetal” cloud server with a 3.5GHz Intel Xeon-Haswell (E3-1270-V3-Quadcore) CPU, 8 GB DDR3 RAM, and a 1 Gbps network connection. The clients run on six baremetal servers in total, each server with 2x 2GHz Intel Xeon-SandyBridge (E5-2620-HexCore) CPUs, 16 GB DD3 RAM, and a 1 Gbps network connection. All clients are hosted in the same data center. All machines run Ubuntu 14.04-64 Linux and Oracle Java (JRE 8, build 1.8.0_77-b03).

To simulate a realistic environment, we conduct experiments in two settings as shown in Table II. A \textit{datacenter} setting, with all components in the same data center (“Amsterdam”), and a \textit{wide-area} setting, where the VICOS server and COS are located together in one data center (“Milan”), and the clients at a remote site (“Amsterdam”).
Table II. Evaluation setting

<table>
<thead>
<tr>
<th>Setting</th>
<th>Clients</th>
<th>VICOS server</th>
<th>Cloud object storage</th>
<th>Latency [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datacenter</td>
<td>Amsterdam</td>
<td>Amsterdam</td>
<td>Amsterdam</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>Wide-area</td>
<td>Amsterdam</td>
<td>Milan</td>
<td>Milan</td>
<td>~ 10</td>
</tr>
</tbody>
</table>

The datacenter setting establishes a best-case baseline due to the very low network latencies (<1 ms). This deployment is not very realistic in terms of the security model because the clients and the storage service are co-located.

The wide-area setting exhibits a moderate network latency (round-trip delay time of ~20 ms) between the two data centers and models the typical case of geographically distributed clients accessing a cloud service with its point-of-access on the same continent but in different countries.

Fig. 3. The experimental setup, with one COSBench controller and many COSBench drivers, accessing the cloud storage service through the VICOS client.

The evaluation is driven by COSBench (Version 0.4.2 – https://github.com/intel-cloud/cosbench), an extensible tool for benchmarking cloud object stores. We have created an adapter to drive VICOS from COSBench, as shown in Fig. 3. COSBench uses a distributed architecture, consisting of multiple drivers, which generate the workload and simulate many clients invoking concurrent operations on a cloud object store, and one controller, which controls the drivers, selects the workload parameters, collects results, and outputs aggregated statistics. In particular, the COSBench setup for VICOS reports the average operation latency, defined as the time that an operation takes from invocation to completion, and the aggregated throughput, defined as the data rate between all clients and the cloud storage service. We present the operation latency and throughput as reported by COSBench. Every reported data point involves read and write operations taken over a period of 30 s after a 30 s warm-up. COSBench supports a closed system model, where clients invoke their next operations only after they have received a response for their previous operation. We have configured COSBench to perform a mix of 50% “read” and 50% “write” operations, implemented by vicos-get and vicos-put, respectively. All reads and writes access different objects (ensuring that all operations are compatible with each other), except for the experiment in Sec. 6.2.4, which evaluates non-compatible operations. In the experiments two configurations are measured:
The native object storage service as a baseline, with direct unprotected access from COSBench to cloud storage, but accessing the cloud storage through the jclouds interface; and

VICOS, running all operations from COSBench through jclouds and the verification protocol.

6.2. Results

6.2.1. Cryptography microbenchmark. In a first experiment we study how different signature implementations affect the computation and network overhead of VICOS. We implemented digital signatures using RSA and DSA with 2048 bit keys, and additionally HMAC-SHA1 with 128 bit keys. The cryptographic algorithms are provided by the SunJCE version 1.8 provider.

We measured the time it takes on a client to sign and verify an INVOKE message using the three signature implementations. Figure 4 shows that RSA signing takes around 5 ms, while verification takes around 220 µs. DSA takes around 4 ms for signing and around 1.5 ms for verification, whereas HMAC signing and verification take less than 20 µs only. Additionally, the resulting signature sizes have a direct effect on the message sizes and the network load. RSA signatures are 256 byte, while DSA signatures are only 40 byte. HMAC reduces the signature size to 20 byte.

We use HMAC-SHA1 as the default implementation of signatures in the remainder of the evaluation. Its operations are much faster than for RSA and DSA signatures, reducing the computation overhead at the client. Moreover, the smaller signature size of HMAC-SHA1 also reduces the network traffic.

Fig. 4. The average time for digital-signature operations (HMAC, RSA, and DSA); note that the y-axis uses log-scale.

6.2.2. Object size. In this experiment we study how the object size affects the latency and the throughput of VICOS. We define a workload with a single client executing read and write operations for objects of size 1 kB, 10 kB, 100 kB, and 1 MB.

Fig. 5 shows that the latency and throughput of VICOS behave very similar to the native system. As expected, we observe that VICOS introduces an overhead that incurs a small cost compared to unprotected access to storage. In particular, for the datacenter setting, VICOS increases latency by an average of 16.2% for read, and 24.0% for write; it decreases throughput by an average of 15.8% for read, and 17.7% for write. We also expect the overhead to decrease with bigger objects. However, we could not find this effect in the datacenter setting: here the overhead remains practically constant, from the small to the large objects. In the wide-area setting, the overhead is approximately the same for the smaller objects (1 kB and 10 kB), but it indeed decreases as the object size grows and disappears at the largest object size (1000 kB).
Interestingly, in the wide-area setting, the relative performance in terms of throughput is reversed between read and write for 1000 kB objects. Whereas read has lower latency and achieves better throughput than write in the other experiments, with smaller objects, this relation is reversed in the right-most data points of Fig. 5. This may be caused by caching data on the cloud object store, which improves read performance for smaller objects, but disappears when larger objects are stored and accessed less frequently than in the datacenter setting.

6.2.3. Number of clients. We also study the scalability of VICOS by increasing the number of clients. The workload uses up to 128 clients (spread uniformly over the six client machines running one COSBench driver each), and 64 objects with a fixed size of 10 kB. One half of the objects are designated for read operations and the other half for write operations, respectively. This division prevents concurrency conflicts among the client operations. As mentioned in Sec. 5.2, we restrict the size of the pending list in the protocol to 128 operations. We do not use a large number of clients because of the underlying assumption that clients trust each other, which might not be realistic in much larger groups.

As Fig. 6 shows, the native system throughput scales linearly until the system is saturated with 64 clients in the data center setting. VICOS follows the same behavior but reaches saturation already with 32 clients. In contrast, in the wide-area setting, VICOS becomes saturated with 8 clients, from where throughput remains almost constant and latency grows. No saturation is evident with the native configuration and up to 128 clients.

The reason for the slower operation in the wide-area experiment is that all requests of the clients are handled by the VICOS server sequentially and thus it becomes a bottleneck of the system. Due to the higher latency in the wide-area setting, operations remain longer in the pending queue. This means more work for the clients and the...
server. Since the active and passive protocol phases are asynchronous, clients may invoke the next operation already before they have completed a previous operation. Hence, the server reaches the limit of the bounded pending queue and stops processing new INVOKE messages until the queue becomes available again. This also limits the throughput of VICOS.

6.2.4. Concurrent operations. Finally, we investigate the effect of conflicting concurrent operations. VICOS aborts an operation if it is not compatible with one of the pending operations (according to Sec. 2.4). In that case the client has to retry later. Protocols like BST [Williams et al. 2009] and ACOP [Cachin and Ohrimenko 2014] are more cautious and abort as soon as two operations do not commute, which occurs more often. Hence, we define ACOP as our baseline for this experiment. The implementation throws an abort exception when a conflict occurs; that causes COSBench to report the operation as failed and to continue immediately with the next operation. At the end of each experiment COSBench reports the overall operation success rate. We expect a higher success rate for VICOS compared to ACOP as already discussed in Sec. 4.2.

To evaluate this behavior we created a workload with sixteen clients each invoking read and write operations over 64 objects with a fixed size of 10 kB. Object accesses are chosen according to “Zipf’s law”, which approximates many types of data series found in natural and social structures. The Zipf distribution is based on a ranking of the elements in a universe and postulates that the frequency of any element is inversely proportional to its rank in the frequency table. Thus, the most “popular” object will occur approximately twice as often as the second most popular one, three times as often as the third most popular and so on. Zipf distributions are often observed when users access websites [Adamic and Huberman 2002], for example. Figure 7 shows the object access rate using four different values for the Zipf factor $\theta$. For $\theta = 0.99$ the access rate for the first object (with the biggest contention) is about 20%, and the least
often accessed object is selected only with probability about 0.3%. With $\theta = 0$ the Zipf distribution corresponds to uniformly random access over the objects.

Since COSBench does not support Zipf distributions by default, we implemented the algorithm of Gray et al. [1994] for generating a Zipf-like access distribution, as also used in YCSB (https://github.com/brianfrankcooper/YCSB). With this workload we cause operations to conflict by progressively increasing $\theta \in \{0, 0.5, 0.75, 0.99\}$. The higher the Zipf factor $\theta$, the more clients concurrently access the same objects and invoke non-compatible operations, causing aborts.

Figure 8 shows the operation success rates for VICOS and for ACOP, using the four different Zipf factors. Recall from Sec. 2.4 and from Table I that a put operation in the KVS interface is always compatible with every preceding operation and never aborts. Therefore, the write operations show 100% success rate for VICOS. For the commuting operations in ACOP, on the other hand, writes are progressively more often aborted with increasing $\theta$. The behavior of reads is similar for VICOS and ACOP because preceding writes cause aborts equally often.

The advantage of VICOS over protocols considering only operation commutativity becomes evident here, in that write operations always succeed, and overall the abort rate is significantly reduced.

6.2.5. Summary. The performance evaluation shows that VICOS achieves its goal of adding consistency and integrity protection while remaining almost transparent to clients using cloud object stores. The cost added over the raw performance is most visible in the datacenter setting, which is not a realistic deployment for the intended applications. Still this overhead remains limited to about 20% for accesses with high
throughput (Sec. 6.2.2). With many clients performing operations concurrently, the extra cost may become noticeable (concretely, for up to about 100 clients, Sec. 6.2.3), and further work is needed for decreasing this. However, recall VICOS is aimed at a group of mutually trusting clients who collaborate on shared data stored in the cloud. It is not our goal to support the full workload of an object-storage cluster and to achieve comparable scalability for 1000s of clients. Furthermore, we note that the VICOS prototype is currently a proof of concept and not product-level code.

7. RELATED WORK

Many previous systems providing data integrity rely on trusted components. Distributed file systems with cryptographic protection provide stronger notions of integrity and consistency than given by VICOS; there are many examples for this, from early research prototypes like FARSITE [Adya et al. 2002] or SiRiUS [Goh et al. 2003] to production file-systems today (e.g., IBM Spectrum Scale, http://www-03.ibm.com/systems/storage/spectrum_scale/). However, they rely on trusted directory services for freshness. Such a trusted coordinator is often missing or considered to be impractical. Iris [Stefanov et al. 2012] relies on a trusted gateway appliance, which mediates all requests between the clients and the untrusted cloud storage. Several recent systems ensure data integrity with the help of trusted hardware, such as CATS [Yumeredefendi and Chase 2007], which offers accountability based on an immutable public publishing medium, or A2M [Chun et al. 2007], which assumes an append-only memory. They all require some form of global synchronization, usually done by the trusted component, for critical metadata to ensure linearizability. In the absence of such communication, as assumed here, they cannot protect consistency and prevent replay attacks.

In CloudProof [Popa et al. 2011], an object-storage protection system with accountable and proof-based data integrity and consistency support, clients may verify the freshness of returned objects with the help of the data owner. Its auditing operation works in epochs and verifies operations on one object only with a certain probability and only at the end of an epoch. Moreover, the clients need to communicate directly with the owner of an object for establishing integrity and consistency.

Cryptographic integrity guarantees are of increasing interest for many diverse domains: Verena [Karapanos et al. 2016], for example, is a recent enhancement for web applications that involve database queries and updates by multiple clients. It targets a patient database holding diagnostic data and treatment information. In contrast to VICOS, however, it relies on a trusted server that supplies fresh values of data objects to clients during every operation and ensures freshness.

The remainder of this section discusses related work without trusted components for synchronization. With only one client, the classic solution for memory checking by Blum et al. [1994] provides data integrity through a hash tree and by storing its root at the client. Many systems have exemplified this approach for remote file systems and for cloud storage (e.g., Athos [Goodrich et al. 2008]).

With authenticated data structures [Naor and Nissim 2000; Martel et al. 2004], the single-writer, multi-reader model of remote storage can be authenticated, assuming there is a trusted and timely way to distribute authenticators from the writer to all readers. In practice, this approach is often taken for software distribution, where new releases are posted to a repository and authenticated by broadcasting hash values of the packages over a mailing list. AIP as introduced in Sec. 3 represents one way to generalize ADS for multiple writers.

In the multi-client model, Mazières and Shasha [2002] have introduced the notion of fork-linearizability and implemented SUNDR [Li et al. 2004], the first system to guarantee fork-linearizable views to all clients. It detects integrity and consistency violations among all clients that become aware of each other's operations. The SUNDR
system uses messages of size $\Omega(n^2)$ for $n$ clients [Cachin et al. 2007], which might be expensive. The SUNDR prototype [Li et al. 2004] description also claims to handle multiple files and directory trees; however, the protocol description and guarantees are stated informally only, so that it remains unclear whether it achieves fork-linearizability under all circumstances.

As mentioned in Sec. 1, several systems have expanded the guarantees of fork-linearizability to different applications [Feldman et al. 2010] and improved the general efficiency of protocols for achieving it [Cachin et al. 2007]. Others have explored aborting operations [Majuntke et al. 2009] or introduced weak fork-linearizability in order to avoid blocking operations. In particular, SPORC [Feldman et al. 2010], FAUST [Cachin et al. 2011b], and Venus [Shraer et al. 2010] sacrifice full linearizability to avoid aborts and blocking, respectively, and achieve weak fork-linearizability instead. The latter is a relaxation of fork-linearizability in which the most recent operation of a client may violate atomicity.

The SPORC system [Feldman et al. 2010] is a groupware collaboration service whose operations may conflict with each other, but can be made to commute by applying a specific technique called “operational transformations.” Through this mechanism, different execution orders still converge to the same state; still SPORC achieves only weak fork-linearizability.

Furthermore, VICOS also reduces the communication overhead compared to past systems considerably, since SUNDR, FAUST, and Venus all use messages of size $\Theta(n)$ or more with $n$ clients, whereas the message size in VICOS does not depend on $n$.

With a focus on high availability and scalability the Depot storage system [Mahajan et al. 2011] ensures fork-causal consistency, where each client observes a causal history of the operations in which it is involved. In the core protocol of Depot clients periodically communicate with each other to exchange data and consistency values; this enables Depot to join forked histories again, and to achieve eventual consistency, which is outside the model of VICOS. Fork-causal consistency relaxes weak fork-linearizability (and transitively also fork-linearizability), but appears more difficult to handle for applications than the strong consistency provided here. Moreover, the model of Depot is complementary to our approach and exploring the techniques of VICOS in the model of Depot would be an interesting option.

The BST protocol [Williams et al. 2009] supports an encrypted remote database hosted by an untrusted server that is accessed by multiple clients. Its consistency checking algorithm allows some commuting client operations to proceed concurrently; COP and ACOP [Cachin and Ohrimenko 2014] extend BST and also guarantee fork-linearizability for arbitrary services run by a Byzantine server, going beyond data storage services, and support wait-freedom for commuting operations. VICOS builds directly on COP, but improves the efficiency by avoiding the local state copies at clients and by reducing the computation and communication overhead. The main advantage is that clients can remain offline between executing operations without stalling the protocol.

8. CONCLUSION

This paper has presented VICOS, a complete system for protecting the integrity and consistency of data outsourced to untrusted commodity cloud object stores. It shows, for the first time, how to realize multi-client integrity protection for generic functions with an authenticated data structure (ADS). Its two-phase protocol structure reduces the communication overhead compared to previous algorithms. VICOS works with commodity cloud storage services and ensures the best possible consistency notion of
Integrity and Consistency for Cloud Object Stores

It supports wait-free client operations and does not require any additional trusted components.

There are several challenges that this paper does not address, which remain open for future work. An interesting question, for instance, is how to recover from an integrity violation. Since we assume only a single untrusted server and that client data resides at the cloud storage service, orthogonal techniques are needed for resilience of the data itself. Another interesting challenge would be to consider malicious clients, as one further step towards a more realistic system. For small groups of clients our system model is realistic, but for groups with hundreds of clients it seems difficult to maintain this assumption. The situation is especially interesting when a client colludes with the malicious server. Finally, the approach of AIP may also be applied to services beyond cloud storage; for example, cloud and NoSQL databases, interactions in a social network, or certificate and key management services.

ACKNOWLEDGMENT

We thank Rüdiger Kapitza, Dieter Sommer and Sören Bleikertz for helpful discussions and the anonymous reviewers for valuable comments.

REFERENCES


ACM Transactions on Privacy and Security, Vol. V, No. N, Article A, Publication date: January YYYY.


ACM Transactions on Privacy and Security, Vol. V, No. N, Article A, Publication date: January YYYY.