Venus: Verification for Untrusted Cloud Storage

Alexander Shraer
Dept. of Electrical Engineering
Technion, Haifa, Israel
shralex@tx.technion.ac.il

Christian Cachin
IBM Research - Zurich
Rüschlikon, Switzerland
cca@zurich.ibm.com

Asaf Cidon*
Dept. of Electrical Engineering
Technion, Haifa, Israel
asaf@cidon.com

Idit Keidar
Dept. of Electrical Engineering
Technion, Haifa, Israel
idish@ee.technion.ac.il

Yan Michalevsky*
Dept. of Electrical Engineering
Technion, Haifa, Israel
ymcrcat@gmail.com

Dani Shaket*
Dept. of Electrical Engineering
Technion, Haifa, Israel
dani.shaket@gmail.com

ABSTRACT

This paper presents Venus, a service for securing user interaction with untrusted cloud storage. Specifically, Venus guarantees integrity and consistency for applications accessing a key-based object store service, without requiring trusted components or changes to the storage provider. Venus completes all operations optimistically, guaranteeing data integrity. It then verifies operation consistency and notifies the application. Whenever either integrity or consistency is violated, Venus alerts the application. We implemented Venus and evaluated it with Amazon S3 commodity storage service. The evaluation shows that it adds no noticeable overhead to storage operations.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; H.3.4 [**Systems and Software**]: Distributed Systems

General Terms

Design, Security, Theory, Verification

Keywords

Cloud storage, integrity, forking semantics, hashing

1. INTRODUCTION

A growing number of cloud providers offer diverse services over the Internet. These include online storage and computing resources, e.g., Amazon Web Services, web application hosts such as Google App Engine, and Software as a Service (SaaS) applications offered by companies like Salesforce.com. Data storage is one of the most prominent cloud applications: individuals store their data online, companies back up local data to the cloud, and many user groups collaborate on data hosted by a remote provider. The ubiquity offered by the Internet and the power to immediately scale and expand the resources available to a client are unique to cloud computing. In addition, the commoditization of cloud computing infrastructure and its pay-on-demand pricing model, coupled with the ability to minimize in-house fixed infrastructure costs, offer a distinct competitive advantage for companies and organizations, large and small.

However, concerns about the trustworthiness of cloud services abound. Clients question the privacy and integrity of their data in the cloud, complain about irregular and unassured availability of SaaS applications, and worry in general about missing quality-of-service guarantees. A number of recent high-profile incidents, such as Amazon S3's silent data corruption[1], a privacy breach in Google Docs[2], and ma.gnolia's data loss[3] rang the alarm bells for cloud users.

Data security is often mentioned as the biggest challenge facing the cloud computing model. This work addresses data integrity and consistency for cloud storage. Two orthogonal concerns are confidentiality, as offered by encryption, and availability, in the sense of resilience and protection against loss (actually, many users consider cloud storage to be more resilient than local storage). Cloud services can be secured following two distinct approaches: from within the cloud infrastructure or from the outside. This work falls in the category of an external security mechanism that can be added transparently to an existing and untrusted service, deployed incrementally, and gives immediate benefits to its clients.

We present *Venus*, short for *VErificatioN for Untrusted Storage*. With Venus, a group of clients accessing a remote storage provider benefits from two guarantees: *integrity* and *consistency*. *Integrity* means that a data object read by any client has previously been written by some client; it protects against simple data modifications by the provider, whether inadvertent or caused by malicious attack. Note that a malicious provider might also try a "replay attack" and answer to a read operation with properly authenticated data from an older version of the object, which has been

*Participated through a project taken in the Networked Software Systems Laboratory.

CCSW'10, October 8, 2010, Chicago, Illinois, USA.

[1] http://developer.amazonwebservices.com/connect/thread.jspa?threadID=22709
[2] http://blogs.wsj.com/digits/2009/03/08/1214/
[3] http://www.wired.com/epicenter/2009/01/magnolia-suffer/

superseded by a newer version. Venus restricts such behavior and guarantees that either the returned data is from the latest write operation to the object, ensuring that clients see atomic operations, or that the provider misbehavior is exposed. This is the *consistency* property of Venus, which allows multiple clients to access the stored data concurrently in a consistent fashion.

Venus notifies the clients whenever it detects a violation of integrity or consistency. Applications may handle this error in a variety of ways, such as switching to another service provider. Venus works transparently with simple object-based cloud storage interfaces, such that clients may continue to work with a commodity storage service of their choice without changing their applications.

During normal operation, clients of cloud storage should not have to communicate with each other. If clients did communicate, they could simply exchange the root value of a hash tree on the stored objects to obtain consistency. This, however, would introduce a prohibitive coordination overhead — clients should be able to execute operations in isolation, when the other clients are disconnected. But without client-to-client communication for every operation, a malicious service could simply ignore write operations by some clients and respond to other clients with outdated data. Previous solutions dealt with the problem using so-called "forking" semantics (in SUNDR [20, 17], and other proposals [7, 19, 4]). These solutions guarantee integrity, and by adding some extra out-of-band communication among the clients can also be used to achieve a related notion of consistency. However, they also incur a major drawback that hampers system availability. Specifically, even when the service functions correctly, all these protocols may sometimes block a client during an operation, requiring the client to wait for another client to finish, and do not guarantee that every client operation successfully completes. It has been shown that this limitation is inherent [6, 5].

Venus eliminates this problem by letting operations finish *optimistically* and establishing consistency later. When the service is correct, all client operations therefore terminate immediately and the service is "wait-free." When an operation returns optimistically, it is called *red*, and Venus guarantees integrity, but not yet consistency. If the storage service is indeed correct, Venus notifies the application later when a red operation is known to be consistent and thereby becomes *green*; in this sense, Venus is eventually consistent [23, 22]. Venus guarantees that the green operations of all clients are consistent, i.e., that they can be ordered in a single sequence of atomic operations. If some red operations are irreconcilable and so may never become green, Venus ensures that every client eventually receives a failure notification.

Venus does not require any additional trusted components and relies only on the clients that are authorized to access the data. Venus allows a dynamic set of potentially disconnected clients. A subset of clients that are frequently online is designated as a *core set*; these clients manage the group membership and help to establish consistency. Venus assumes that clients are correct or may crash silently, but otherwise follow their specification, and that a majority of the clients in the core set is correct. The storage service usually functions properly, but may become subject to attacks or behave arbitrarily. Venus is asynchronous and never violates consistency or integrity due to timeouts, but relies on some synchrony assumptions for liveness. Clients may occasionally communicate with each other by email. Since this is conducted in the background, independently of storage operations, and only if a client suspects that the storage service is faulty, it does not affect the performance of Venus.

Our implementation of Venus is comprised of a *client-side library* and a *verifier*. The client-side library overrides the interface of the storage service, extending it with eventual consistency and failure notifications. The verifier brokers consistency information and can be added to the storage service in a modular way; typically it will also run in the cloud, hosted by the same untrusted service that provides the storage. Internally, the verifier and the storage service might be replicated for fault tolerance and high availability. Note that using replication within the cloud does not solve the problem addressed by Venus, since from the client's perspective, the entire cloud is a single trust domain. We stress that Venus does not trust the verifier any more than the storage service — the two entities may collude arbitrarily against the clients, and separating them simply supports a layered implementation with commodity providers. Of course, the verifier could be run by a trusted third party, but it would be a much stronger assumption and existing protocols suffice for integrity and consistency in this model [2].

We have implemented Venus and deployed it using the commodity Amazon S3 cloud storage service[4]. Venus requires an additional message exchange between client and verifier for each operation, in addition to accessing the raw storage. We report on experiments using Venus connected to S3 and with a verifier deployed either on a remote server or on the same LAN as the clients. We compare the performance of storage access using Venus to that of the raw S3 service. Both the latency and the throughput of Venus closely match the performance of the raw S3 service. Specifically, when the verifier is deployed on the local LAN, Venus' performance is identical to that of S3. When the verifier is deployed remotely, Venus adds a small overhead to latency compared to S3 (corresponding to one round of additional communication with the verifier) and achieves the same throughput. We have also tested Venus' capability to detect service misbehavior and present logs from such an experiment, where the clients communicate with each other and detect that the cloud storage provider has violated consistency (as simulated).

Contributions.

Our results demonstrate that data integrity and consistency for remote storage accessed by multiple clients can be obtained without significant overhead, no additional trusted components, and seamlessly integrated with the normal operations. Specifically, Venus is the first practical decentralized algorithm that

- verifies cryptographic integrity and consistency of remotely stored data accessed by multiple clients without introducing trusted components,
- does not involve client-to-client coordination or introduce extra communication on the critical path of normal operations,
- provides simple semantics to clients, lets operations execute optimistically, but guarantees that either all

[4] `http://aws.amazon.com/s3/`

operations eventually become consistent, or that every client is informed about the service failure, and
- is practically implemented on top of a commodity cloud storage service.

Venus may secure a variety of applications that currently use cloud storage, such as online collaboration, Internet backup, and document archiving. No less important is that Venus enables many applications that require verifiable guarantees and do not blindly trust a service provider to be deployed in the cloud.

Organization.

The remainder of the paper is organized as follows: Section 2 discusses related work. Section 3 presents the design of Venus, and Section 4 defines its semantics. The protocol for clients and the verifier is given in Section 5. Section 6 describes our implementation of Venus, and finally, Section 7 presents its evaluation.

2. RELATED WORK

Data integrity on untrusted storage accessed by a single client with small trusted memory can be protected by storing the root of a hash tree locally [2]. Systems applying this approach to outsourced file systems and to cloud storage have also been demonstrated [10, 11].

In cryptographic storage systems with multiple clients, such "root hashes" are additionally signed; TDB [18], SiRiUS [9], and Plutus [15] are some representative examples implementing this method. In order to ensure freshness, the root hashes must be propagated by components that are at least partially trusted, however. Going beyond ensuring the integrity of data that is actually read from an untrusted service by a single client, recent work by Juels and Kaliski [14] and by Ateniese et al. [1] introduces protocols for assuring the client that it can retrieve its data in the future, with high probability. Unlike Venus, this work does not guarantee consistency for multiple clients accessing the data concurrently.

Several recent systems provide integrity using trusted components, which cannot be subverted by intrusions. A prominent system of this kind is CATS [26], which provides storage accountability based on an immutable public bulletin board available to the service and to all clients. Another proposal is A2M [8], which utilizes a trusted append-only memory (realized in hardware or software) to ensure atomic semantics. Venus uses client signatures on the data, but no trusted components.

A separate line of work provides so-called forking semantics [20], which are weaker than conventional atomic semantics, but these systems do not require any trusted components whatsoever. SUNDR [17], Cachin et al. [7, 4] and Majuntke et al. [19] propose storage systems of this kind that ensure forking consistency semantics. They guarantee that after a single consistency violation by the service, the views seen by two inconsistent clients can never again converge. The main drawbacks of these systems lie, first, in the difficulty of understanding forking semantics and exploiting them in applications, and, second, in their monolithic design, which integrates storage operations with the consistency mechanism. Hence, it is difficult to use these approaches for securing practical cloud storage.

Furthermore, the systems with forking semantics mentioned so far [17, 7, 4] may block reading clients when a read-write conflict occurs [6, 5]. In such situations, readers cannot progress until the writer completes its operation, which is problematic, especially if the writer crashes. Majuntke et al. [19] and Williams et al. [24] provide fork-consistency and guarantee system-wide progress but their algorithms may abort some conflicting operations. Going beyond fork-consistent protocols designed for untrusted storage, the system of Williams et al. [24] provides an untrusted database server and supports transactions. In contrast, Venus never blocks a client operation when the storage service is correct, and every client may proceed independently of other clients and complete every operation. Venus provides more intuitive system semantics, whereby operations complete optimistically before their consistency is verified. In the absence of failures, every client operation is eventually marked as green, and Venus ensures that all clients observe a single sequence of green operations.

FAUST [5] implements the notion of weak fork-linearizability, which allows client operations to complete optimistically, as in Venus. It also provides notifications to clients, but they are different and less intuitive — FAUST issues *stability* notifications, where each notification includes a vector indicating the level of synchronization that a client has with every other client. This stability notion is not transitive and requires every client to explicitly track the other clients in the system and to assess their relation to the data accessed by an operation. FAUST is therefore not easily amenable to dynamic changes in the set of clients. Furthermore, it is unclear how clients can rely on FAUST stability notifications in a useful manner; global consistency in FAUST (among all clients) is guaranteed only if no client ever crashes. FAUST does not work with commodity storage; like other proposals, it integrates storage operations with the consistency mechanism, and it does not allow multiple clients to modify the same object, which is the usual semantics of commodity storage services.

In contrast, indications in Venus simply specify the last operation of the client that has been verified to be globally consistent, which is easy to integrate with an application. Venus eliminates the need for clients to track each other, and enables dynamic client changes. Unlike the previous protocols [5, 7, 19], Venus allows all clients to modify the same shared object. Most importantly, the design of Venus is modular, so that it can be deployed with a commodity storage service.

Orthogonal to this work, many storage systems have been proposed that internally use replication across several nodes to tolerate a fraction of corrupted nodes (e.g., [12] and references therein). For instance, HAIL [3] is a recent system that relies replicated storage servers internally, of which at least a majority must be correct at any time. It combines data replication with a method that gives proofs of retrievability to the clients. But a storage service employing replication within its cloud infrastructure does not solve the problem addressed by Venus — from the perspective of the client, the cloud service is still a single trust domain.

3. SYSTEM MODEL

Figure 1 depicts our system model, which includes a *storage service*, a generic commodity online service for storing and retrieving objects of arbitrary size, a *verifier*, which implements our consistency and verification functions and multiple *clients*. The storage service is used as is, without any

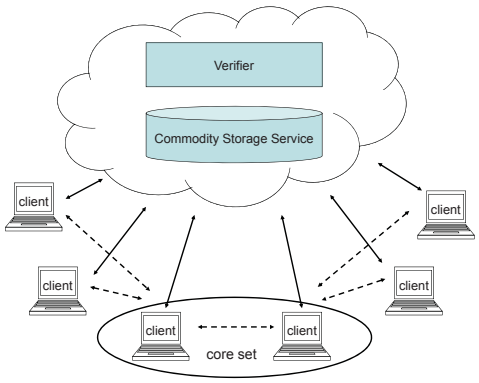


Figure 1: Venus Architecture.

modification. Usually the storage service and the verifier are hosted in the same cloud and will be correct; but they may become faulty or corrupted by an adversary, and they may collude together against the clients.

There are an arbitrary number of clients, which are subject to crash failures. Clients may be connected intermittently and are frequently offline. The *core set* of clients is a publicly known subset of the clients with a special role. These clients help detect consistency and failures (Section 5.4) and manage client membership (Section 5.6); to this end, clients occasionally communicate directly with clients from the core set. A quorum of the core set of clients must not crash (but may also be offline temporarily). Note that clients of cloud services, and especially users of cloud storage, do not operate continuously. Hence, clients should not depend on other clients for liveness of their operations. Indeed, every operation executed by a client in Venus proceeds independently of other clients and promptly completes, even if all other clients are offline.

Clients in Venus are honest and do not deviate from their specification (except for crashing). Note that tolerating malicious clients does not make a lot of sense, because every client may write to the shared storage. From the perspective of the correct clients, the worst potential damage by another client is to simply overwrite the storage with bogus information. Venus, just like commodity cloud storage, cannot not perform application-specific validation of written data.

Venus clients are admitted by a member of the core set, as determined by the same access-control policy as the one used at the commodity storage interface. Clients are identified by a signature public key and an email address, bound together with a self-signed certificate. Every client knows initially at least the public keys of all clients in the core set.

Messages between clients and the verifier or the storage service are sent over reliable point-to-point channels. Client-to-client communication is conducted using digitally signed email messages; this allows clients to go temporarily offline or to operate behind firewalls and NATs. Clients rarely communicate directly with each other.

The storage service is assumed to have an interface for writing and reading data objects. The *write* operation takes the identifier *obj* of the object and some data x as parameters and returns an acknowledgment. The *read* operation expects an object identifier *obj* and returns the data stored in the object. After a new object is successfully stored, clients are able to read it within a bounded period of time, though perhaps not immediately. We assume that this bound is known; in practice, it can be obtained dynamically[5]. The assumption that such time threshold exists reflects clients' expectation from any usable storage service. Inability to meet this expectation (e.g., due to an internal partition) can be perceived as a failure of the storage provider as far as clients are concerned. Venus makes several attempts to read the object, until this time bound is exceeded, at which time a failure notification is issued to clients.

4. VENUS INTERFACE AND SEMANTICS

Venus overrides the $write(obj, x)$ and $read(obj)$ operations for accessing an object identified by *obj* in the interface of the storage service. Venus does not allow partial updates of an object, the value x overwrites the value stored previously. If the object does not exist yet, it is created. For simplicity of presentation, we assume that each client executes operations sequentially.

Venus extends the return values of write and read operations by a local timestamp t, which increasing monotonically with the sequence of operations executed by the client. An operation o always completes optimistically, without waiting for other clients to complete their operations; at this point, we say that o is *red*, which means that the integrity of the operation has been checked, but its consistency is yet unverified.

A weak form of consistency is nevertheless guaranteed for all operations that become red. Namely, they ensure *causal consistency* [13], which means intuitively that all operations are consistent with respect to potential causality [16]. For example, a client never reads two causally related updates in the wrong order. In addition, it guarantees that a read operation never returns an outdated value, if the reader was already influenced by a more recent update. Causality has proven to be important in various applications, such as various collaborative tools and Web 2.0 applications [21, 25]. Although usually necessary for applications, causality is often insufficient. For example, it does not rule out replay attacks or prevent two clients from reading two different versions of an object.

Venus provides an asynchronous callback interface to a client, which issues periodic consistency and failure notifications. A consistency notification specifies a timestamp t that denotes the most recent *green* operation of the client, using the timestamp returned by operations. All operations of the client up to this operations have been verified to to be consistent and are also green. Intuitively, all clients observe the green operations in the same order. More precisely, Venus ensures that there exists a global sequence of operations, including at least the green operations of all clients, in which the green operations appear according to their order of execution. Moreover, this sequence is *legal*, in the sense that every read operation returns the value written by the last write that precedes the read in the sequence, or an empty value if no such write exists. Note that the sequence might include some red operations, in addition to the green ones. This may happen, for instance, when a client starts

[5]Amazon guarantees that S3 objects can be read immediately after they are created: `http://aws.typepad.com/aws/2009/12/aws-importexport-goes-global.html`

to write and crashes during the operation, and a green read operation returns the written value.

Failure notifications indicate that the storage service or the verifier has violated its specification. Venus guarantees that every complete operation eventually becomes green, unless the client executing it crashes, or a failure is detected.

5. PROTOCOL DESCRIPTION

Section 5.1 describes the interaction of Venus clients with the storage service. Section 5.2 describes *versions*, used by Venus to check consistency of operations. Section 5.3 presents the protocol between the clients and the verifier. Section 5.4 describes how clients collect information from other clients (either through the verifier or using client-to-client communication), and use it for eventual consistency and failure detection. Section 5.5 describes optimizations.

For simplicity, we first describe the protocol for a fixed set of clients $C_1, \ldots, C_n$, and relax this assumption later in Section 5.6. The algorithm uses several timeout parameters, which are introduced in Figure 2. We have formally proven that Venus provides the properties defined in Section 4; these proofs are omitted due to space limitations.

In what follows, we distinguish between objects provided by Venus and which can be read or written by applications, and objects which Venus creates on storage. The former are simply called objects, while the latter are called *low-level objects*. Every update made by the application to an object *obj* managed with Venus creates a new low-level object at the storage service with a unique identifier, denoted p_x in the description below, and the verifier maintains a pointer to the last such update for every object managed by Venus. Clients periodically garbage-collect such low-level objects (see also Section 5.5).

R	Number of times an operation is retried on the storage service.
t_{dummy}	Frequency of dummy-read operations.
t_{send}	Time since last version observed from another client, before that client is contacted directly.
$t_{receive}$	Frequency of checking for new messages from other clients.

Figure 2: Venus timeout parameters.

5.1 Overview of read and write operations

The protocol treats all objects in the same way; we therefore omit the object identifier in the sequel.

The general flow of read and write operations is presented in Figure 3. When a *write*(x) operation is invoked at a client C_i to update the object, the client calculates h_x, a cryptographic hash of x, and writes x to the storage service, creating a new low-level object with a unique path p_x, chosen by the client-side library. Using p_x as a handle, the written data can later be retrieved from storage. Notice that p_x identifies the low-level object created for this update, and it is different from the object identifier exported by Venus, which is not sent to storage. After the low-level write completes, C_i sends a SUBMIT message to the verifier including p_x and h_x, informing it about the write operation. C_i must wait before sending the SUBMIT message, since if C_i crashes before x is successfully stored, p_x would not be a valid handle and read operations receiving p_x from the verifier would fail when trying to retrieve x from the storage service. The verifier orders all SUBMIT messages, creating a global sequence $\mathcal{H}$ of operations.

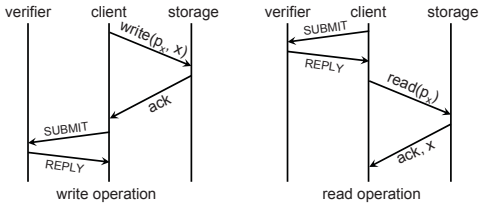


Figure 3: Operation flow.

When a *read* operation is invoked, the client first sends a SUBMIT message to the verifier, in order to retrieve a handle corresponding to the latest update written to the object. The verifier responds with a REPLY message including p_x and h_x from the latest update. The reader then contacts the storage service and reads the low-level object identified by p_x. In most cases, the data will be returned by the storage service. The reader then checks the integrity of the data by computing its hash and comparing it to h_x; if they are equal, it returns the data to the application. If the storage provider responds that no low-level object corresponds to p_x, the client re-executes the read. If the correct data can still not be read after R repetitions, the client announces a failure. Similarly, failure is announced if hashing the returned data does not result in h_x. Updates follow the same pattern: if the storage does not successfully complete the operation after R attempts, then the client considers it faulty.

Since the verifier might be faulty, a client must verify the integrity of all information sent by the verifier in REPLY messages. To this end, clients sign all information they send in SUBMIT messages. A more challenging problem, which we address in the next section, is verifying that p_x and h_x returned by the verifier correspond to the latest write operation, and in general, that the verifier orders the operations correctly.

5.2 From timestamps to versions

In order to check that the verifier constructs a correct sequence $\mathcal{H}$ of operations, our protocol requires the verifier to supply the *context* of each operation in the REPLY. The context of an operation o is the prefix of $\mathcal{H}$ up to o, as determined by the client that executes o. This information can be compactly represented using *versions* as follows.

Every operation executed by a client C_i has a local timestamp, returned to the application when the operation completes. The timestamp of the first operation is 1 and it is incremented for each subsequent operation. We denote the timestamp of an operation o by $ts(o)$. Before C_i completes o, it determines a vector-clock value $vc(o)$ representing the context of o; the j-th entry in $vc(o)$ contains the timestamp of the latest operation executed by client C_j in o's context.

In order to verify that operations are consistent with respect to each other, more information about the context of each operation is needed. Specifically, the context is compactly represented by a version, as in previous works [20, 7, 5]. A *version*(o) is a pair composed of the vector-clock *version*$(o).vc$, which is identical to $vc(o)$, and a second vector, *version*$(o).vh$, where the i-th entry contains a cryptographic hash of the prefix of $\mathcal{H}$ up to o. This hash is

computed by iteratively hashing all operations in the sequence with a cryptographic collision-resistant hash function. Suppose that o_j is the last operation of C_j in C_i's context, i.e., $version(o).vc[j] = ts(o_j)$. Then, the j-th entry in $version(o).vh$ contains a representation (in the form of a hash value) of the prefix of $\mathcal{H}$ up to o_j. Client C_i calculates $version(o).vh$ during the execution of o according to the information provided by the verifier in the REPLY message. Thus, if the verifier follows its protocol, then $version(o).vh[j]$ is equal to $version(o_j).vh[j]$.

For simplicity, we sometimes write $vc(o)$ and $vh(o)$ for $version(o).vc$ and $version(o).vh$, respectively. We define the following order (similarly to [20, 7, 5]), which determines whether o could have appeared before another operation o' in the same legal sequence of operations:

Order on versions: $version(o) \leq version(o')$ whenever both of the following conditions hold:

1. $vc(o) \leq vc(o')$, i.e., for every k, $vc(o)[k] \leq vc(o')[k]$.
2. For every k such that $vc(o)[k] = vc(o')[k]$, it holds that $vh(o)[k] = vh(o')[k]$.

The first condition checks that the context of o' includes at least all operations that appear in the context of o. Suppose that o_k is the last operation of C_k appearing both in the context of o and in that of o'. In this case, the second condition verifies that the prefix of $\mathcal{H}$ up to o_k is the same in the contexts of o and o'. We say that two versions are *comparable* when one of them is smaller than or equal to the other. The existence of incomparable versions indicates a fault of the verifier.

5.3 Operation details

Each client maintains a version corresponding to its most recently completed operation o_{prev}. Moreover, if o_{prev} is a read operation, the client keeps p_{prev} and h_{prev} retrieved by o_{prev} from the verifier. Note that client C_i does not know context and version of its current operation when it sends the SUBMIT message, as it only computes them after receiving the REPLY. Therefore, it sends the version of o_{prev} with the SUBMIT message its next operation to the verifier.

When sending the SUBMIT message for a READ operation o, C_i encloses a representation of o (including the timestamp $ts(o)$), the version o_{prev} of its previous operation as well as a signature on $vh(o_{prev})[i]$. Such a signature is called a *proof* and authenticates the prefix of C_i's context of o_{prev}. If o is a write operation, the message also includes the tuple $(p_x, h_x, ts(o))$, where p_x is the handle and h_x is the hash of the data already written to the storage provider. Otherwise, if o is a read operation, and o_{prev} was also a read, the message includes $(p_{prev}, h_{prev}, ts(o_{prev}))$. All information in the SUBMIT message is signed (except for the proof, which is a signature by itself).

Recall that the verifier constructs the global sequence $\mathcal{H}$ of operations. It maintains an array *Ver*, in which the j-th entry holds the last version received from client C_j. Moreover, the verifier keeps the index of the client from which the maximal version was received in a variable c; in other words, $Ver[c]$ is the maximal version in *Ver*. We denote the operation with version $Ver[c]$ by o_c. The verifier also maintains a list *Pending*, containing the operations that follow o_c in $\mathcal{H}$. Hence, operations appear in *Pending* according to the order in which the verifier received them from clients (in SUBMIT messages). Furthermore, a variable *Proofs* contains an array of proofs from SUBMIT messages. Using this array, clients will be able to verify their consistency with C_j up to C_j's previous operation, before they agree to include C_j's next operation in their context.

Finally, the verifier stores an array *Paths* containing the tuple $(p_x, h_x, ts(o))$ received most recently from every client. Notice that if the last operation of a client C_j is a *write*, then this tuple is included in the SUBMIT message and the verifier updates $Paths[j]$ when it receives the SUBMIT. On the other hand, the SUBMIT message of a *read* operation does not contain the handle and the hash; the verifier updates $Paths[j]$ only when it receives the next SUBMIT message from C_j. The verifier processes every SUBMIT message atomically, updating all state variables together, before processing the next SUBMIT message.

After processing a SUBMIT message, the verifier sends a REPLY message that includes c, $version(o_c)$, *Pending*, *Proofs* (only those entries in *Proofs* which correspond to clients executing operations in *Pending*), and for a read operation also a tuple (p_x, h_x, t_x) with a handle, hash, and timestamp as follows. If there are write operations in *Pending*, then the verifier takes (p_x, h_x, t_x) from the entry in *Paths* corresponding to the client executing the last write in *Pending*. Otherwise, if there are no writes in *Pending*, then it uses the tuple (p_x, h_x, t_x) stored in $Paths[c]$.

When C_i receives the REPLY message for its operation o, it verifies the signatures on all information in the message, and then performs the following checks:

1. The maximal version sent by the verifier, $version(o_c)$, is at least as big as the version corresponding to C_i's previous operation, $version(o_{prev})$.
2. The timestamp $ts(o_{prev})$ of C_i's previous operation is equal to $vc(o_c)[i]$, as o_{prev} should be the last operation that appears in the context of o_c.
3. If o is a read operation, then (p_x, h_x, t_x) indeed corresponds to the last write operation in *Pending*, or to o_c if there are no write operations in *Pending*. This can be checked by comparing t_x to the timestamp of the appropriate operation in *Pending* or to $ts(o_c)$, respectively.

Next, C_i computes $version(o)$, by invoking the function shown in Figure 4, to represent o's context based on the prefix of the history up to o_c (represented by $version(o_c)$), and on the sequence of operators in *Pending*. The following additional checks require traversing *Pending*, and are therefore performed during the computation of $version(o)$, which iterates over all operations in *Pending*:

4. There is at most one operation of every client in *Pending*, and no operation of C_i, that is, the verifier does not include too many operations in *Pending*.
5. For every operation o by client C_j in *Pending*, the timestamp $ts(o)$ is equal to $vc(o_c)[j] + 1$, that is, o is indeed the next operation executed by C_j after the one appearing in the context of o_c.
6. For every client C_j that has an operation in *Pending*, $Proofs[j]$ is a valid signature by C_j on $vh(o_c)[j]$. That is, the context of o_c includes and properly extends the context of the previous operation of C_j, as represented by the hash $vh(o_c)[j]$ and the signature $Proofs[j]$.

```
1: function compute-version-and-check-pending(o)
2:    (vc, vh) ← version(o_c)
3:    histHash ← vh[c]
4:    for q = 1, ..., |Pending| :          // traverse pending ops
5:       let C_j be the client executing Pending[q]
6:       vc[j] ← vc[j] + 1
7:       histHash ← hash(histHash||Pending[q])
8:       vh[j] ← histHash
9:       perform checks 4, 5, and 6 (see text below)
10:   version(o) = (vc, vh)
11:   return version(o)
```

Figure 4: Computing the version of an operation.

If one of the checks fails, the application is notified and a failure message is sent to the core set clients, as described in Section 5.4.

5.4 Detecting consistency and failures

An application of Venus registers for two types of callback notifications: consistency notifications, which indicate that some operations have become green and are known to be consistent, and failure notifications, issued when a failure of the storage service or the verifier has been detected. Below we describe the additional mechanisms employed by the clients for issuing such notifications, including client-to-client communication.

Each client C_i maintains an array $CVer$. For every client C_j in the core set, $CVer[j]$ holds the biggest version of C_j known to C_i. The entries in $CVer$ might be outdated, for instance, when C_i has been offline for a while, and more importantly, $CVer[j]$ might not correspond to an operation actually executed by C_j, as we explain next. Together with each entry of $CVer$, the client keeps the local time of its last update to the entry.

Every time a client C_i completes an operation o, it calculates $version(o)$ and stores it in $CVer[i]$. To decide whether its own operations are globally consistent, C_i must also collect versions from other clients. More precisely, it needs to obtain the versions from a majority quorum of clients in the core set. Usually, these versions arrive via the verifier, but they can also be obtained using direct client-to-client communication.

To obtain another client's version via the verifier, C_i piggybacks a VERSION-REQUEST message with every SUBMIT message that it sends. The VERSION-REQUEST message includes the identifier k of some client in the core set. In response, the verifier includes $Version[k]$ with the REPLY message. When C_i receives the REPLY message, it updates $CVer[k]$ if the received version is bigger than the old one (of course, the signature on the received version must be verified first). Whenever C_i executes an operation, it requests the version of another client from the core set in the VERSION-REQUEST message, going in round-robin over all clients in the core set. When no application-invoked operations are in progress, the client also periodically (every t_{dummy} time units) issues a dummy-read operation, to which it also attaches VERSION-REQUEST messages. The dummy-read operations are identical to application-invoked reads, except that they do not access the storage service after processing the REPLY message. A dummy-read operation invoked by C_i causes an update to $Version[i]$ at the verifier, even though no operation is invoked by the application at C_i. Thus, clients that repeatedly request the version of C_i from the verifier see an increasing sequence of versions of C_i.

It is possible, however, that C_k goes offline or crashes, in which case C_i will not see a new version from C_k and will not update $CVer[k]$. Moreover, a faulty verifier could be hiding C_k's new versions from C_i. To client C_i these two situations look the same. In order to make progress faced with this dilemma, C_i contacts C_k directly whenever $CVer[k]$ does not change for a predefined time period t_{send}. More precisely, C_i sends the maximal version in $CVer$ to C_k, asking C_k to respond with a similar message. When C_k is online, it checks for new messages from other clients every $t_{receive}$ time units, and thus, if C_k has not permanently crashed, it will eventually receive this message and check that the version is comparable to the maximum version in its array $CVer$. If no errors are found, C_k responds to C_i with the maximal version from $CVer$, as demonstrated in Figure 5(a). Notice that this maximal version does not necessarily correspond to an operation executed by C_i. All client-to-client messages use email and are digitally signed to prevent attacks from the network.

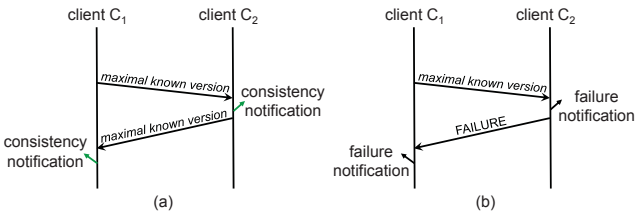


Figure 5: Consistency checks using client-to-client communication. In (a) the checks pass, which leads to a response message and consistency notifications. In (b) one of the checks fails and C_2 broadcasts a failure message.

When a client C_i receives a version directly from C_k, it makes sure the received version is comparable with the maximal version in its array $CVer$. If the received version is bigger than $CVer[k]$, then C_i updates the entry.

Whenever an entry in $CVer$ is updated, C_i checks whether additional operations become green, which can be determined from $CVer$ as explained next. If this is the case, Venus notifies the application and outputs the timestamp of the latest green operation. To check if an operation o becomes green, C_i invokes the function in Figure 6, which computes a *consistency set* $\mathcal{C}(o)$ of o. If $\mathcal{C}(o)$ contains a majority quorum of the clients in the core set, the function returns *green*, indicating that o is now known to be consistent.

```
1: function check-consistency(o)
2:    C(o) ← ∅
3:    for each client C_k in the core set:
4:       if CVer[k].vc[i] ≥ ts(o) then
5:          add k to C(o)
6:    if C(o) contains a quorum of the core set then
7:       return green
8:    else
9:       return red
```

Figure 6: Checking whether o is green.

C_i starts with the latest application-invoked (non-dummy) red operation o, going over its red operations in reverse order of their execution, until the first application-invoked red operation o is encountered that becomes green. If such an

operation o is found, C_i notifies the application that all operations with timestamps smaller than or equal to $ts(o)$ are now green.

If at any point a check made by the client fails, it broadcasts a failure message to all core set clients; when receiving such message for the first time, a core set client forwards this message to all other core set clients. When detecting a failure or receiving a failure message, a client notifies its application and ceases to execute application-invoked and dummy operations. After becoming aware of a failure, a core set client responds with a failure message to any received version message, as demonstrated in Figure 5(b).

5.5 Optimizations and garbage collection

Access to the storage service consumes the bulk of execution time for every operation. Since this time cannot be reduced by our application, we focus on overlapping as much of the computation as possible with the access to storage.

For a read operation, as soon as a REPLY message is received, the client immediately starts reading from the storage service, and concurrently makes all checks required to complete its current operation. In addition, the client prepares (and signs) the information about the current operation that will be submitted with its next operation (notice that this information does not depend on the data returned by the storage service).

A write operation is more difficult to parallelize, since a SUBMIT message cannot be sent to the verifier before the write to the storage service completes. This is due to the possibility that a SUBMIT message reaches the verifier but the writer crashes before the data is successfully written to the storage service, creating a dangling pointer at the verifier. If this happens, no later read operation will be able to complete successfully.

We avoid this problem by proceeding with the write optimistically, without changing the state of the client or verifier. Specifically, while the client awaits the completion of its write to the storage, it sends a DUMMY-SUBMIT message to the verifier, as shown in Figure 7. Unlike a normal SUBMIT, this message is empty and thus cannot be misused by the verifier, e.g., by presenting it to a reader as in the scenario described above. When receiving a DUMMY-SUBMIT message, the verifier responds with a REPLY message identical to the one it would send for a real SUBMIT message (notice that a REPLY message for a write operation does not depend on the contents of the SUBMIT message). The writer then optimistically makes all necessary checks, calculations and signatures. When storing the data is complete, the client sends a SUBMIT message to the verifier. If the REPLY message has not changed, pre-computed information can be used, and otherwise, the client re-executes the checks and computations for the newly received information.

Venus creates a new low-level object at the storage provider for every write operation of the application. In fact, this is exactly how updates are implemented by most cloud storage providers, which do not distinguish between overwriting an existing object and creating a new one. This creates the need for garbage collection. We have observed, however, that with Amazon S3 the cost of storing multiple low-level objects for a long period of time is typically much smaller than the cost of actually uploading them (which is anyway necessary for updates), thus eager garbage collection will not significantly reduce storage costs. In Venus, each client periodically garbage-collects low-level objects on storage corresponding to its outdated updates.

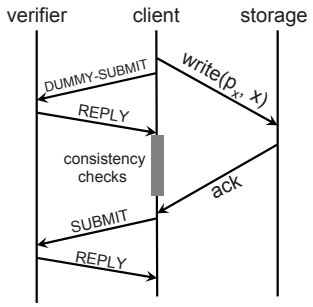


Figure 7: Speculative write execution.

5.6 Joining the system

We have described Venus for a static set of clients so far, but in fact, Venus supports dynamic client joins. In order to allow for client joins, clients must have globally unique identifiers. In our implementation these are their unique email addresses. All arrays maintained by the clients and by the verifier, including the vector clock and the vector of hashes in versions, are now associative arrays, mapping a client identifier to the corresponding value. Clients may also leave Venus silently but the system keeps their entries in versions.

The verifier must not accept requests from clients for which it does not have a public key signed by some client in the core set. As mentioned in Section 3, every client wishing to join the system knows the core set of clients and their public keys. To join the system, a new client C_i sends a JOIN message, including its public key, to some client in the core set; if the client does not get a response it periodically repeats the process until it gets a successful response. When receiving a JOIN request from C_i, a client C_j in the core set checks whether C_i can be permitted access to the service using the externally defined access policy, which permits a client to access Venus if and only if the client may also access the object at the storage service. If access to C_i is granted, C_j still needs to verify that C_i controls the public key in the JOIN message. To this end, C_j asks the joining client to sign a nonce under the supplied public key, as shown in Figure 8.

If the signature returned by C_j is valid, then C_j signs C_i's public key and sends it to the verifier. After the verifier has

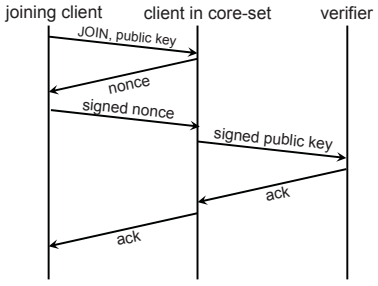


Figure 8: Flow of a join operation.

acknowledged its receipt, C_j sends a final acknowledgment to C_i, and from this time on, C_i may invoke read and write operations in Venus.

The verifier informs a client C_i about clients that are yet unknown to C_i, by including their signed public keys in REPLY messages to C_i. In order to conclude what information C_i is missing, the verifier inspects $version(o_{prev})$ received from C_i in the SUBMIT message, where it can see which client identifiers correspond to values in the associative arrays. A client receiving a REPLY message extracts all public keys from the message and verifies that the signature on each key was made by a client from the core set. Then, it processes the REPLY message as usual. If at any time some information is received from the verifier, but a public key needed to verify this information is missing, then C_i concludes that the verifier is faulty and notifies its application and the other clients accordingly.

6. IMPLEMENTATION

We implemented Venus in Python 2.6.3, with Amazon S3 as the storage service. Clients communicate with S3 using HTTP. Communication with the verifier uses direct TCP connections or HTTP connections; the latter allow for simpler traversal of firewalls.

Client-to-client communication is implemented by automated emails. This allows our system to handle offline clients, as well as clients behind firewalls or NATs. Clients communicate with their email provider using SMTP and IMAP for sending and receiving emails, respectively. Clients are identified by their email addresses.

For signatures we used GnuPG. Specifically, we used 1024-bit DSA signatures. Each client has a local key-ring where it stores the public keys corresponding to clients in our system. Initially the key-ring stores only the keys of the clients in the core set, and additional keys are added as they are received from the verifier, signed by some client in the core set. We use SHA-1 for hashing.

Venus does not access the versioning support of Amazon S3, which was announced only recently, and relies on the basic key-value store functionality.

To evaluate how Venus detects service violations of the storage service and the verifier, we simulated some attacks. Here we demonstrate one such scenario, where we simulate a "split-brain" attack by the verifier, in a system with two clients. Specifically, the verifier conceals operations of each client from the other one. Figure 9 shows the logs of both clients as generated by the Venus client-side library. We observe that one email exchange suffices to detect the inconsistency.

7. EVALUATION

We report on measurements obtained with Venus for clients deployed at the Technion (Haifa, Israel), Amazon S3 with the US Standard Region as the storage service, and with the verifier deployed at MIT (Cambridge, USA) and locally at the Technion.

The clients in our experiments run on two IBM 8677 Blade Center chassis, each with 14 JS20 PPC64 blades. We dedicate 25 blades to the clients, each blade having 2 PPC970FX cores (2.2 GHz), 4GB of RAM and 2 BroadCom BCM5704S NICs. When deployed locally, the verifier runs on a separate HS21 XM blade, Intel QuadCore Xeon E5420 with 2.5GHz, 16GB of RAM and two BroadCom NetXtreme II BCM5708S NICs. In this setting the verifier is connected to the clients by a 1Gb Ethernet.

When run remotely at MIT, the verifier is hosted on a shared Intel Xeon CPU 2.40GHz machine with 2GB RAM. In this case, clients contact the verifier using HTTP, for tunneling through a firewall, and the requests reach the Venus verifier redirected by a CGI script on a web server.

All machines run the Linux 2.6 operating system.

7.1 Operation latency

We examine the overhead Venus introduces for a client executing operations, compared to direct, unverified access to S3, which we denote here by "raw S3."

Figure 10 shows the average operation latency for a single client executing operations (since there is a single client in this experiment, operations become green immediately upon completing). The latencies are shown for raw S3, with the verifier in the same LAN as the client at the Technion, and with the remote verifier at MIT. Each measurement is an average of the latencies of 300 operations, with the 95% confidence intervals shown. We measure the average latency for different sizes of the data being read or written, namely 1KB, 10KB, 100KB and 1000KB.

Figure 10 shows that the latency for accessing raw S3 is very high, in the orders of seconds. Many users have previously reported similar measurements[6,7]. The large confidence intervals for 1000KB stem from a high variance in the latency (also previously reported by S3 users) of accessing big objects on S3. The variance did not decrease when an average of 1000 operations was taken.

The graphs show that the overhead of using Venus compared to using Amazon S3 directly depends on the location of the verifier. When the verifier is local, the overhead is negligible. When it is located far from the clients, the overhead is constant (450-550 ms.) for all measured data sizes. It stems from one two-way message exchange between the client and verifier, which takes two round-trip times in practice, one for establishing a TCP connection and another one for the message itself. Although we designed the verifier and the clients to support persistent HTTP connections, we found that the connection remained open only between each client and a local proxy, and was closed and re-opened between intermediate nodes in the message route. We suspect the redirecting web server does not support keeping HTTP connections open.

We next measure the operation latency with multiple clients and a local verifier. Specifically, we run 10 clients, 3 of which are the core set. Half of the clients perform read operations, and half of them perform writes; each client executes 50 operations. The size of the data in this experiment is 4KB. Figure 11 shows the average time for an operation to complete, i.e., to become red, as well as the time until it becomes green, with t_{dummy} set to 3 sec., or to 5 sec. Client-to-client communication was disabled for these experiments.

One can observe that as the time between user-invoked operations increases, the average latency of green notifications initially grows as well, because versions advance at a slower rate, until the dummy-read mechanism kicks in and

[6] http://bob.pythonmac.org/archives/2006/12/06/cachefly-vs-amazon-s3/

[7] http://developer.amazonwebservices.com/connect/message.jspa?messageID=93072

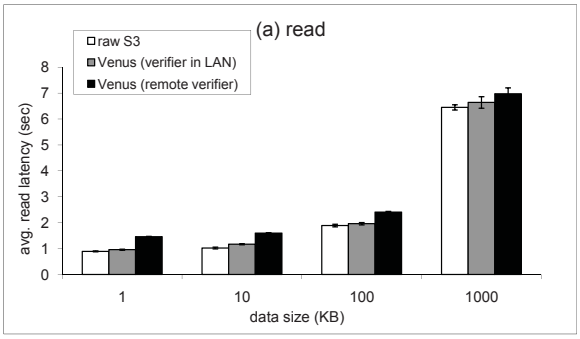
Log of Client #1: venusclient1@gmail.com

09:26:38: initializing client venusclient1@gmail.com
09:26:43: executing dummy-read with <REQUEST-VERSION, venusclient2@gmail.com>
-----"----: no update to CVersions[venusclient2@gmail.com]
09:26:45: received email from client venusclient2@gmail.com. Signature OK
-----"----: failure detected: venusclient2@gmail.com sent an incomparable version
-----"----: notifying other clients and shutting down...

Log of Client #2: venusclient2@gmail.com

09:26:30: initializing client venusclient2@gmail.com
09:26:35: executing dummy-read with <REQUEST-VERSION, venusclient1@gmail.com>
-----"----: no update to CVersions[venusclient1@gmail.com]
09:26:40: executing dummy-read with <REQUEST-VERSION, venusclient1@gmail.com>
-----"----: no update to CVersions[venusclient1@gmail.com]
-----"----: sending version to client venusclient1@gmail.com, requesting response
09:26:45: executing dummy-read with <REQUEST-VERSION, venusclient1@gmail.com>
-----"----: no update to CVersions[venusclient1@gmail.com]
09:26:49: received email from client venusclient1@gmail.com. Signature OK
-----"----: failure reported by client venusclient1@gmail.com
-----"----: notifying other clients and shutting down...

Figure 9: Client logs from detecting a simulated "split-brain" attack, where the verifier hides each client's operations from the other clients. System parameters were set to $t_{dummy} = 5sec.$, $t_{send} = 10sec.$, and $t_{receive} = 5sec.$ There are two clients in the system, which also form the core set. After 10 seconds, client #2 does not observe a new version corresponding to client #1 and contacts it directly. Client #1 receives this email, and finds the version in the email to be incomparable to its own latest version, as its own version does not reflect any operations by client #2. The client replies reporting of an error, both clients notify their applications and halt.



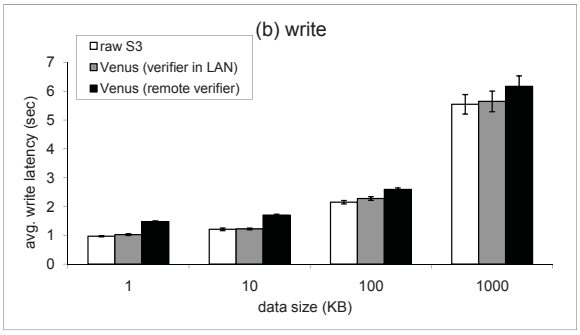


Figure 10: Average latency of a read and write operations, with 95% confidence intervals. The overhead is negligible when the verifier is the same LAN as the client. The overhead for WAN is constant.

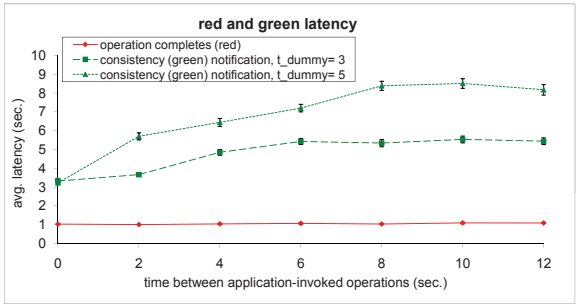


Figure 11: Average latency for operations with multiple clients to become red and green respectively.

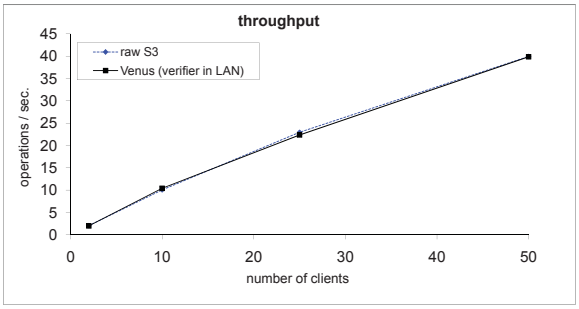


Figure 12: Average throughput with multiple clients.

ensures steady progress. Of course the time it takes for an operation to complete, i.e., to become red, is not affected by the frequency of invocations.

7.2 Verifier

Knowing that the overhead of our algorithm at the client-side is small, we proceed to test the verifier's scalability and throughput. Since our goal here is to test the verifier under high load, we perform this stress test with a synthetic multi-client program, which simulates many clients to the server. The simulated clients only do as much as is needed to flood the verifier with plausible requests.

Amazon S3 does not support pipelining HTTP operation requests, and thus, an operation of a client on S3 has to end before that client can invoke another operation. Consequently, the throughput for clients accessing raw S3 can be expected to be the number of client threads divided by the average operation latency. In order to avoid side effects caused by contention for processing and I/O resources, we do not run more 2 client threads per each of our 25 dual-core machines, and therefore measure throughput with up to 50 client threads. As Venus clients access Amazon S3 for each application-invoked operation, our throughput cannot exceed that of raw S3, for a given number of clients. Our measurements show that the throughput of Venus is almost identical to that of raw S3, as can be seen in Figure 12.

8. CONCLUSIONS

In this paper we presented Venus, a practical service that guarantees integrity and consistency to users of untrusted cloud storage. Venus can be deployed transparently with commodity online storage and does not require any additional trusted components. Unlike previous solutions, Venus offers simple semantics and never aborts or blocks client operations when the storage is correct. We implemented Venus and evaluated it with Amazon S3. The evaluation demonstrates that Venus has insignificant overhead and can therefore be used by applications that require cryptographic integrity and consistency guarantees while using online cloud storage.

Acknowledgments

We thank Maxim Gurevich and Jean-Philippe Martin for helpful discussions and the anonymous reviewers for valuable comments.

Alexander Shraer was supported by an Eshkol Fellowship from the Israeli Ministry of Science.

This work has also been supported in part by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II.

9. REFERENCES

[1] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proc. 14th ACM Conference on Computer and Communications Security (CCS)*, pages 598–609, 2007.

[2] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12:225–244, 1994.

[3] K. D. Bowers, A. Juels, and A. Oprea. HAIL: A high-availability and integrity layer for cloud storage. In *Proc. 16th ACM Conference on Computer and Communications Security (CCS)*, pages 187–198, 2009.

[4] C. Cachin and M. Geisler. Integrity protection for revision control. In M. Abdalla and D. Pointcheval, editors, *Proc. Applied Cryptography and Network Security (ACNS)*, volume 5536 of *Lecture Notes in Computer Science*, pages 382–399, 2009.

[5] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 494–503, 2009.

[6] C. Cachin, I. Keidar, and A. Shraer. Fork sequential consistency is blocking. *Inf. Process. Lett.*, 109(7):360–364, 2009.

[7] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 129–138, 2007.

[8] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proc. 21st ACM Symposium on Operating System Principles (SOSP)*, pages 189–204, 2007.

[9] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2003.

[10] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In T.-C. Wu et al., editors, *Proc. 11th Information Security Conference (ISC)*, volume 5222 of *Lecture Notes in Computer Science*, pages 80–96, 2008.

[11] A. Heitzmann, B. Palazzi, C. Papamanthou, and R. Tamassia. Efficient integrity checking of untrusted network storage. In *Proc. Workshop on Storage Security and Survivability (StorageSS)*, pages 43–54, 2008.

[12] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead Byzantine fault-tolerant storage. In *SOSP*, pages 73–86, 2007.

[13] P. W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enchance concurrency in distributed shared memories. In *Proc. 10th Intl. Conference on Distributed Computing Systems (ICDCS)*, pages 302–309, 1990.

[14] A. Juels and B. S. Kaliski. PORs: Proofs of retrievability for large files. In *Proc. 14th ACM Conference on Computer and Communications Security (CCS)*, pages 584–597, 2007.

[15] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[16] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[17] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. 6th Symp. Operating Systems Design and Implementation (OSDI)*, pages 121–136, 2004.

[18] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proc. 4th Symp. Operating Systems Design and Implementation (OSDI)*, pages 135–150, 2000.

[19] M. Majuntke, D. Dobre, M. Serafini, and N. Suri. Abortable fork-linearizable storage. In T. F. Abdelzaher, M. Raynal, and N. Santoro, editors, *Proc. 13th Conference on Principles of Distributed Systems (OPODIS)*, volume 5923 of *Lecture Notes in Computer Science*, pages 255–269, 2009.

[20] D. Mazières and D. Shasha. Building secure file systems out of Byzantine storage. In *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 108–117, 2002.

[21] A. Milani. Causal consistency in static and dynamic distributed systems. PhD Thesis, "La Sapienza" Università di Roma, 2006.

[22] M. Serafini, D. Dobre, M. Majuntke, P. Bokor, and N. Suri. Eventually linearizable shared objects. In *Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 95–104, 2010.

[23] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. 15th ACM Symposium on Operating System Principles (SOSP)*, pages 172–182, 1995.

[24] P. Williams, R. Sion, and D. Shasha. The blind stone tablet: Outsourcing durability to untrusted parties. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2009.

[25] J. Yang, H. Wang, N. Gu, Y. Liu, C. Wang, and Q. Zhang. Lock-free consistency control for Web 2.0 applications. In *Proc. 17th Intl. Conference on World Wide Web (WWW)*, pages 725–734, 2008.

[26] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. *ACM Transactions on Storage*, 3(3), 2007.