Consistency as a Service: Auditing Cloud Consistency

Qin Liu, Guojun Wang, *Member, IEEE,* and Jie Wu, *Fellow, IEEE*

***Abstract*—Cloud storage services have become commercially popular due to their overwhelming advantages. To provide ubiquitous always-on access, a cloud service provider (CSP) maintains multiple replicas for each piece of data on geographically distributed servers. A key problem of using the replication technique in clouds is that it is very expensive to achieve strong consistency on a worldwide scale. In this paper, we first present a novel consistency as a service (CaaS) model, which consists of a large data cloud and multiple small audit clouds. In the CaaS model, a data cloud is maintained by a CSP, and a group of users that constitute an audit cloud can verify whether the data cloud provides the promised level of consistency or not. We propose a two-level auditing architecture, which only requires a loosely synchronized clock in the audit cloud. Then, we design algorithms to quantify the severity of violations with two metrics: the commonality of violations, and the staleness of the value of a read. Finally, we devise a heuristic auditing strategy (HAS) to reveal as many violations as possible. Extensive experiments were performed using a combination of simulations and real cloud deployments to validate HAS.**

***Index Terms*—Cloud storage, consistency as a service (CaaS), two-level auditing, heuristic auditing strategy (HAS).**

I. Introduction

CLOUD computing has become commercially popular, as it promises to guarantee scalability, elasticity, and high availability at a low cost [1], [2]. Guided by the trend of the everything-as-a-service (XaaS) model, data storages, virtualized infrastructure, virtualized platforms, as well as software and applications are being provided and consumed as services in the cloud. Cloud storage services can be regarded as a typical service in cloud computing, which involves the delivery of data storage as a service, including database-like services and network attached storage, often billed on a utility computing basis, e.g., per gigabyte per month. Examples include Amazon SimpleDB[1], Microsoft Azure storage[2], and so on. By using the cloud storage services, the customers can access data stored in a cloud anytime and anywhere, using any device, without caring about a large amount of capital investment when deploying the underlying hardware infrastructures.

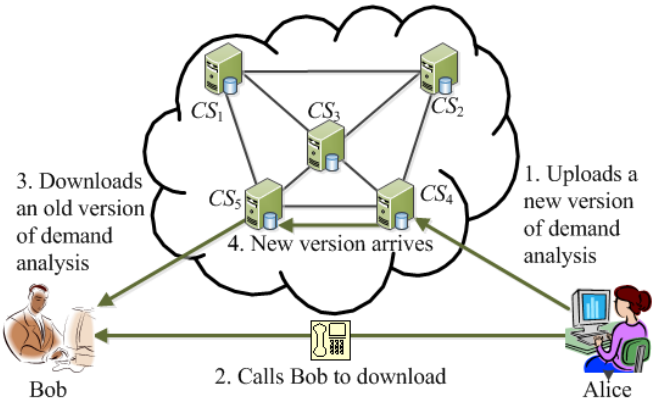

Fig. 1. An application that requires causal consistency.

To meet the promise of ubiquitous 24/7 access, the cloud service provider (CSP) stores data replicas on multiple geographically distributed servers. A key problem of using the replication technique in clouds is that it is very expensive to achieve strong consistency on a worldwide scale, where a user is ensured to see the latest updates. Actually, mandated by the CAP principle[3], many CSPs (e.g., Amazon S3) only ensure weak consistency, such as eventual consistency, for performance and high availability, where a user can read stale data for a period of time. The domain name system (DNS) is one of the most popular applications that implement eventual consistency. Updates to a name will not be visible immediately, but all clients are ensured to see them eventually.

However, eventual consistency is not a catholicon for all applications. Especially for the interactive applications, stronger consistency assurance is of increasing importance. Consider the following scenario as shown in Fig. 1. Suppose that Alice and Bob are cooperating on a project using a cloud storage service, where all of the related data is replicated to five cloud servers, $CS_1, \ldots, CS_5$. After uploading a new version of the requirement analysis to a CS_4, Alice calls Bob to download the latest version for integrated design. Here, after Alice calls Bob, the causal relationship [5] is established between Alice's update and Bob's read. Therefore, the cloud should provide causal consistency, which ensures that Alice's update is committed to all of the replicas before Bob's read. If the

Manuscript received June 25, 2013; revised October 9, 2013. The special issue guest editors coordinating the review of this paper and approving it for publication were G. Martinez, R. Campbell, and J. Alcaraz Calero.

Q. Liu is with the School of Information Science and Engineering, Central South University, Changsha, Hunan Province, P. R. China, 410083. Q. Liu is also with the College of Information Science and Engineering, Hunan University, Changsha, Hunan Province, P. R. China, 410082 (e-mail: gracelq628@126.com).

G. Wang, the corresponding author, is with the School of Information Science and Engineering, Central South University, Changsha, Hunan Province, P.R. China, 410083 (e-mail: csgjwang@mail.csu.edu.cn).

J. Wu is with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA (e-mail: jiewu@temple.edu).

Digital Object Identifier 10.1109/TNSM.2013.122613.130411

[1]aws.amazon.com/simpledb/

[2]www.windowsazure.com

[3]CAP principle [3], [4] states that any shared data system can provide only two of the following three properties: consistency, availability, and partition tolerance. Since partitions are inevitable in wide-area networks, distributed systems should trade consistency for availability.

cloud provides only eventual consistency, then Bob is allowed to access an old version of the requirement analysis from CS_5. In this case, the integrated design that is based on an old version may not satisfy the real requirements of customers.

Actually, different applications have different consistency requirements. For example, mail services need monotonic-read consistency and read-your-write consistency, but social network services need causal consistency [6]. In cloud storage, consistency not only determines correctness but also the actual cost per transaction. In this paper, we present a novel *consistency as a service* (CaaS) model for this situation. The CaaS model consists of a large *data cloud* and multiple small *audit clouds*. The data cloud is maintained by a CSP, and an audit cloud consists of a group of users that cooperate on a job, e.g., a document or a project. A service level agreement (SLA) will be engaged between the data cloud and the audit cloud, which will stipulate what level of consistency the data cloud should provide, and how much (monetary or otherwise) will be charged if the data cloud violates the SLA.

The implementation of the data cloud is opaque to all users due to the virtualization technique. Thus, it is hard for the users to verify whether each replica in the data cloud is the latest one or not. Inspired by the solution in [7], we allow the users in the audit cloud to verify cloud consistency by analyzing a trace of interactive operations. Unlike their work, we do not require a global clock among all users for total ordering of operations. A loosely synchronized clock is suitable for our solution. Specifically, we require each user to maintain a logical vector [8] for partial ordering of operations, and we adopt a two-level auditing structure: each user can perform *local auditing* independently with a local trace of operations; periodically, an auditor is elected from the audit cloud to perform *global auditing* with a global trace of operations. Local auditing focuses on monotonic-read and read-your-write consistencies, which can be performed by a light-weight online algorithm. Global auditing focuses on causal consistency, which is performed by constructing a directed graph. If the constructed graph is a directed acyclic graph (DAG), we claim that causal consistency is preserved. We quantify the severity of violations by two metrics for the CaaS model: commonality of violations and staleness of the value of a read, as in [9]. Finally, we propose a *heuristic auditing strategy* (HAS) which adds appropriate reads to reveal as many violations as possible. Our key contributions are as follows:

1) We present a novel consistency as a service (CaaS) model, where a group of users that constitute an audit cloud can verify whether the data cloud provides the promised level of consistency or not.
2) We propose a two-level auditing structure, which only requires a loosely synchronized clock for ordering operations in an audit cloud.
3) We design algorithms to quantify the severity of violations with different metrics.
4) We devise a heuristic auditing strategy (HAS) to reveal as many violations as possible. Extensive experiments were performed using a combination of simulations and real cloud deployments to validate HAS.

The remainder of this paper is organized as follows: We introduce related work in Section II and present preliminaries in Section III. We describe verification algorithms for the two-level auditing structure in Section IV, before we provide algorithms to quantify the severity of violations in Section V. After we propose a heuristic auditing strategy to reveal as many violations as possible in Section VI, we conduct experiments to validate the heuristic auditing strategy in Section VII. Finally, we provide additional discussion in Section VIII and conclude this paper in Section IX.

II. Related Work

A cloud is essentially a large-scale distributed system where each piece of data is replicated on multiple geographically-distributed servers to achieve high availability and high performance. Thus, we first review the consistency models in distributed systems. Ref. [10], as a standard textbook, proposed two classes of consistency models: data-centric consistency and client-centric consistency. Data-centric consistency model considers the internal state of a storage system, i.e., how updates flow through the system and what guarantees the system can provide with respect to updates. However, to a customer, it really does not matter whether or not a storage system internally contains any stale copies. As long as no stale data is observed from the client's point of view, the customer is satisfied. Therefore, client-centric consistency model concentrates on what specific customers want, i.e., how the customers observe data updates. Their work also describes different levels of consistency in distributed systems, from strict consistency to weak consistency. High consistency implies high cost and reduced availability. Ref. [11] states that strict consistency is never needed in practice, and is even considered harmful. In reality, mandated by the CAP protocol [3], [4], many distributed systems sacrifice strict consistency for high availability.

Then, we review the work on achieving different levels of consistency in a cloud. Ref. [12] investigated the consistency properties provided by commercial clouds and made several useful observations. Existing commercial clouds usually restrict strong consistency guarantees to small datasets (Google's MegaStore and Microsoft's SQL Data Services), or provide only eventual consistency (Amazon's simpleDB and Google's BigTable). Ref. [13] described several solutions to achieve different levels of consistency while deploying database applications on Amanzon S3. In Ref. [14], the consistency requirements vary over time depending on actual availability of the data, and the authors provide techniques that make the system dynamically adapt to the consistency level by monitoring the state of the data. Ref. [15] proposed a novel consistency model that allows it to automatically adjust the consistency levels for different semantic data.

Finally, we review the work on verifying the levels of consistency provided by the CSPs from the users' point of view. Existing solutions can be classified into trace-based verifications [7], [9] and benchmark-based verifications [16]–[19]. Trace-based verifications focus on three consistency semantics: safety, regularity, and atomicity, which are proposed by Lamport [20], and extended by Aiyer et al. [21]. A register is safe if a read that is not concurrent with any write returns the value of the most recent write, and a read that is concurrent with a write can return any value. A register is regular if a

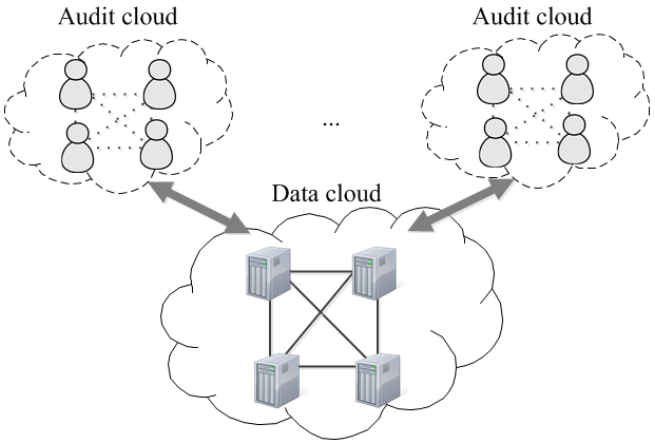

Fig. 2. Consistency as a service model.

read that is not concurrent with any write returns the value of the most recent write, and a read that is concurrent with a write returns either the value of the most recent write, or the value of the concurrent write. A register is atomic if every read returns the value of the most recent write. Misra [22] is the first to present an algorithm for verifying whether the trace on a read/write register is atomic. Following his work, Ref. [7] proposed offline algorithms for verifying whether a key-value storage system has safety, regularity, and atomicity properties by constructing a directed graph. Ref. [9] proposed an online verification algorithm by using the GK algorithm [23], and used different metrics to quantify the severity of violations. The main weakness of the existing trace-based verifications is that a global clock is required among all users. Our solution belongs to trace-based verifications. However, we focus on different consistency semantics in commercial cloud systems, where a loosely synchronized clock is suitable for our solution.

Benchmark-based verifications focus on benchmarking staleness in a storage system. Both [16] and [17] evaluated consistency in Amazon's S3, but showed different results. Ref. [16] used only one user to read data in the experiments, and showed that few inconsistencies exist in S3. Ref. [17] used multiple geographically-distributed users to read data, and found that S3 frequently violates monotonic-read consistency. The results of [17] justify our two-level auditing structure. Ref. [18] presents a client-centric benchmarking methodology for understanding eventual consistency in distributed key-value storage systems. Ref. [19] assessed Amazon, Google, and Microsoft's offerings, and showed that, in Amazon S3, consistency was sacrificed and only a weak consistency level known as, eventual consistency, was achieved.

III. Preliminaries

In this section, we first illustrate the consistency as a service (CaaS) model. Then, we describe the structure of *the user operation table* (UOT), with which each user records his operations. Finally, we provide an overview of the two-level auditing structure and related definitions.

A. Consistency as a Service (CaaS) Model

As shown in Fig. 2, the CaaS model consists of a *data cloud* and multiple *audit clouds*. The data cloud, maintained by the cloud service provider (CSP), is a key-value data storage system [24], where each piece of data is identified by a unique key. To provide always-on services, the CSP replicates all of the data on multiple geographically distributed cloud servers.

An audit cloud consists of a group of users that cooperate on a job, e.g., a document or a program. We assume that each user in the audit cloud is identified by a unique ID. Before outsourcing the job to the data cloud, the audit cloud and the data cloud will engage in a service level agreement (SLA), which stipulates the promised level of consistency that should be provided by the data cloud. The audit cloud exists to verify whether the data cloud violates the SLA or not, and to quantify the severity of violations.

In our system, a two-level auditing model is adopted: each user records his operations in a user operation table (UOT), which is referred to as a local trace of operations in this paper. *Local auditing* can be performed independently by each user with his own UOT; periodically, an auditor is elected from the audit cloud. In this case, all other users will send their UOTs to the auditor, which will perform *global auditing* with a global trace of operations. We simply let each user become an auditor in turn, and we will provide a more comprehensive solution in Section VIII. The dotted line in the audit cloud means that users are loosely connected. That is, users will communicate to exchange messages after executing a set of reads or writes, rather than communicating immediately after executing every operation. Once two users finish communicating, a causal relationship on their operations is established.

B. User Operation Table (UOT)

Each user maintains a UOT for recording local operations. Each record in the UOT is described by three elements: *operation*, *logical vector*, and *physical vector*. While issuing an operation, a user will record this operation, as well as his current logical vector and physical vector, in his UOT.

Each operation op is either a write $W(K, a)$ or a read $R(K, a)$, where $W(K, a)$ means writing the value a to data that is identified by key K, and $R(K, a)$ means reading data that is identified by key K and whose value is a. As in [7], we call $W(K, a)$ as $R(K, a)$'s *dictating write*, and $R(K, a)$ as $W(K, a)$'s *dictated read*. We assume that the value of each write is unique. This is achieved by letting a user attach his ID, and current vectors to the value of write. Therefore, we have the following properties: (1) A read must have a unique dictating write. A write may have zero or more dictated reads. (2) From the value of a read, we can know the logical and physical vectors of its dictating write.

Each user will maintain a logical vector and a physical vector to track the logical and physical time when an operation happens, resepectively. Suppose that there are N users in the audit cloud. A logical/physical vector is a vector of N logical/physical clocks, one clock per user, sorted in ascending order of user ID. For a user with ID_i where $1 \leq i \leq N$, his logical vector is $< LC_1, LC_2, \ldots, LC_N >$, where LC_i is his logical clock, and LC_j is the latest logical clock of user j to his best knowledge; his physical vector is $< PC_1, PC_2, \ldots, PC_N >$, where PC_i is his physical clock, and PC_j is the latest physical clock of user j, to the best of his knowledge.

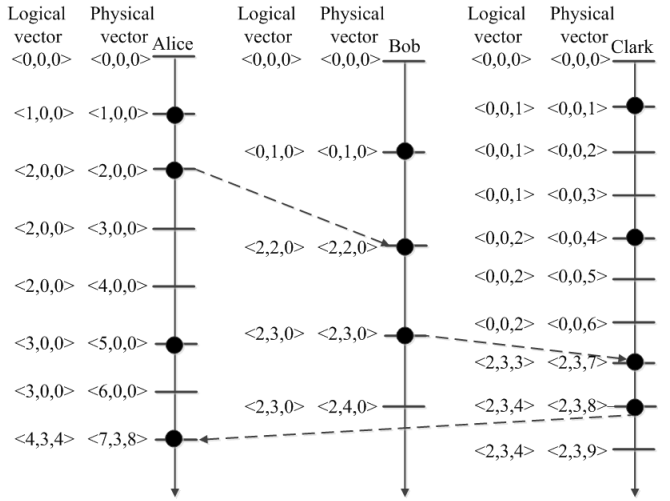

Fig. 3. The update process of logical vector and physical vector. A black solid circle denotes an event (read/write/send message/receive message), and the arrows from top to bottom denote the increase of physical time.

The logical vector is updated via the *vector clocks* algorithm [8]. The physical vector is updated in the same way as the logical vector, except that the user's physical clock keeps increasing as time passes, no matter whether an *event* (read/write/send message/receive message) happens or not. The update process is as follows: All clocks are initialized with zero (for two vectors); The user increases his own physical clock in the physical vector continuously, and increases his own logical clock in the logical vector by one only when an event happens; Two vectors will be sent along with the message being sent. When a user receives a message, he updates *each element* in his vector with the maximum of the value in his own vector and the value in the received vector (for two vectors).

To illustrate, let us suppose that there are three users in the audit cloud, Alice, Bob, and Clark, where $ID_{Alice} < ID_{Bob} < ID_{Clark}$. Each user may update the vectors as shown in Fig. 3. If the first event for Alice is $W(K, a)$, the first record in Alice's UOT is $[W(K, a), < 1, 0, 0 >, < 1, 0, 0 >]$. This means that Alice writes value a to data identified by key K when both her physical and logical clocks are 1. Furthermore, when this event happens, she has no information about other users' clocks, which are thus set with the initial value 0. Note that, since there is no global time in the audit cloud, the number of clock ticks in each user's physical clock may be different, e.g., in Fig. 3, when Alice's physical clock passed seven clock ticks, Bob's physical clock passed only four ticks.

C. Overview of Two-Level Auditing Structure

Vogels [12] investigated several consistency models provided by commercial cloud systems. Following their work, we provide a two-level auditing structure for the CaaS model. At the first level, each user independently performs local auditing with his own UOT. The following consistencies (also referred to as local consistencies) should be verified at this level:

Monotonic-read consistency. *If a process reads the value of data K, any successive reads on data K by that process will always return that same value or a more recent value.*

Read-your-write consistency. *The effect of a write by a process on data K will always be seen by a successive read on data K by the same process.*

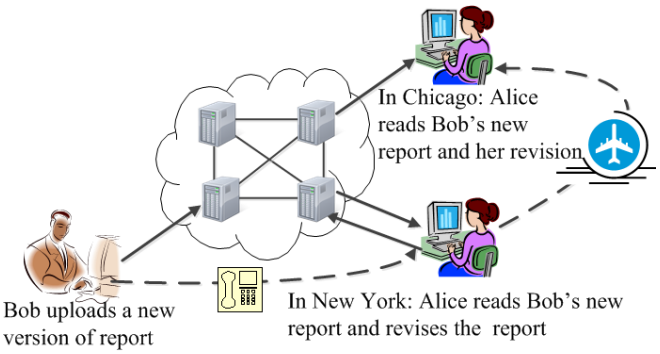

Fig. 4. An application that has different consistency requirements.

Algorithm 1 Local consistency auditing

```
Initial UOT with ∅
while issue an operation op do
  if op = W(a) then
    record W(a) in UOT
  if op = r(a) then
    W(b) ∈ UOT is the last write
    if W(a) → W(b) then
      Read-your-write consistency is violated
    R(c) ∈ UOT is the last read
    if W(a) → W(c) then
      Monotonic-read consistency is violated
    record r(a) in UOT
```

Intuitively, monotonic-read consistency requires that a user must read either a newer value or the same value, and read-your-write consistency requires that a user always reads his latest updates. To illustrate, let us consider the example in Fig. 4. Suppose that Alice often commutes between New York and Chicago to work, and the CSP maintains two replicas on cloud servers in New York and Chicago, respectively, to provide high availability. In Fig. 4, after reading Bob's new report and revising this report in New York, Alice moves to Chicago. Monotonic-read consistency requires that, in Chicago, Alice must read Bob's new version, i.e., the last update she ever saw in New York must have been propagated to the server in Chicago. Read-your-write consistency requires that, in Chicago, Alice must read her revision for the new report, i.e., her own last update issued in New York must have been propagated to the server in Chicago. The above models can be combined. The users can choose a subset of consistency models for their applications.

At the second level, an auditor can perform global auditing after obtaining a global trace of all users' operations. At this level, the following consistency (also referred to as global consistency in this paper) should be verified:

Causal consistency. *Writes that are causally related must be seen by all processes in the same order. Concurrent writes*

may be seen in a different order on different machines.

Here, a casual relationship between events, denoted as $\rightsquigarrow$, can be defined by the following rules [5]:

- If e_1 and e_2 are two events in a process, and e_1 happens before e_2, then $e_1 \rightsquigarrow e_2$.
- If e_1 is e_2's dictating write and e_2 is e_1's dictated read, then $e_1 \rightsquigarrow e_2$.
- If e_1 is a send event and e_2 is a receive event, then $e_1 \rightsquigarrow e_2$.
- If $e_1 \rightsquigarrow e_2$ and $e_2 \rightsquigarrow e_3$, then $e_1 \rightsquigarrow e_3$.

If $e_1 \not\rightsquigarrow e_2$ and $e_2 \not\rightsquigarrow e_1$, then e_1 is concurrent with e_2, denoted by $e_1||e_2$. The happen-before relationship, denoted as $\rightarrow$, can be defined with logical vector as follows: If $LV(e_1) < LV(e_2)$[4], then $e_1 \rightarrow e_2$, where $LV(e_i)$ denotes the logical vector of operation e_i for $i \in \{1, 2\}$.

In the application scenario in Fig. 4, after uploading a new version of the report to the data cloud, Bob calls Alice, asking her to download it. After the call, Bob's update and Alice's read are causally related. Therefore, causal consistency requires that Alice must read Bob's new report.

IV. Verification of Consistency Properties

In this section, we first provide the algorithms for the two-level auditing structure for the CaaS model, and then analyze their effectiveness. Finally, we illustrate how to perform a garbage collection on UOTs to save space. Since the accesses of data with different keys are independent of each other, a user can group operations by key and then verify whether each group satisfies the promised level of consistency. In the remainder of this paper, we abbreviate read operations with $R(a)$ and write operations with $W(a)$.

A. Local Consistency Auditing

Local consistency auditing is an online algorithm (Alg. 1). In Alg. 1, each user will record all of his operations in his UOT. While issuing a read operation, the user will perform local consistency auditing independently.

Let $R(a)$ denote a user's current read whose dictating write is $W(a)$, $W(b)$ denote the last write in the UOT, and $R(c)$ denote the last read in the UOT whose dictating write is $W(c)$. Read-your-write consistency is violated if $W(a)$ happens before $W(b)$, and monotonic-read consistency is violated if $W(a)$ happens before $W(c)$. Note that, from the value of a read, we can know the logical vector and physical vector of its dictating write. Therefore, we can order the dictating writes by their logical vectors.

B. Global Consistency Auditing

Global consistency auditing is an offline algorithm (Alg. 2). Periodically, an auditor will be elected from the audit cloud to perform global consistency auditing. In this case, all other users will send their UOTs to the auditor for obtaining a global trace of operations. After executing global auditing, the auditor will send auditing results as well as its vectors to all other users. Given the auditor's vectors, each user will know other users' latest clocks up to global auditing.

[4]Let $LV(e_i)_j$ denote user j's logical clock in $LV(e_i)$. $LV(e_1) < LV(e_2)$ if $\forall j[LV(e_1)_j \leq LV(e_2)_j] \wedge \exists j[LV(e_1)_j < LV(e_2)_j]$.

Algorithm 2 Global consistency auditing

Each operation in the global trace is denoted by a vertex
for any two operations op_1 and op_2 **do**
  if $op_1 \rightarrow op_2$ **then**
    A time edge is added from op_1 to op_2
  if $op_1 = W(a)$, $op_2 = R(a)$, and two operations come from different users **then**
    A data edge is added from op_1 to op_2
  if $op_1 = W(a)$, $op_2 = W(b)$, two operations come from different users, and $W(a)$ is on the route from $W(b)$ to $R(b)$ **then**
    A causal edge is added from op_1 to op_2
Check whether the graph is a DAG by topological sorting

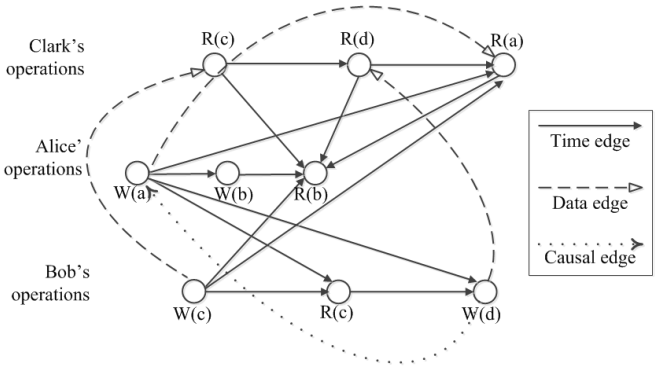

Fig. 5. Sample graph constructed with Alg. 2.

Inspired by the solution in [7], we verify consistency by constructing a directed graph based on the global trace. We claim that causal consistency is preserved if and only if the constructed graph is a directed acyclic graph (DAG). In Alg. 2, each operation is denoted by a vertex. Then, three kinds of directed edges are added by the following rules:

1) Time edge. For operation op_1 and op_2, if $op_1 \rightarrow op_2$, then a directed edge is added from op_1 to op_2.
2) Data edge. For operations $R(a)$ and $W(a)$ that come from different users, a directed edge is added from $W(a)$ to $R(a)$.
3) Causal edge. For operations $W(a)$ and $W(b)$ that come from different users, if $W(a)$ is on the route from $W(b)$ to $R(b)$, then a directed edge is added from $W(a)$ to $W(b)$.

Take the sample UOTs in Table I as an example. The graph constructed with Alg. 2 is shown in Fig. 5. This graph is not a DAG. From Table I, we know that $W(a) \rightarrow W(d)$, as $LV(W(a)) < LV(W(d))$. Ideally, a user should first read the value of a and then d. However, user Clark first reads the value of d and then a, violating causal consistency.

To determine whether a directed graph is a DAG or not, we can perform topological sorting [25] on the graph. Any DAG has at least one topological ordering, and the time complexity of topological sorting is $O(V+E)$, where V is the number of vertexes and E is the number of edges in the graph. To reduce the running time of topological sorting, we can modify Alg.

TABLE I
SAMPLE UOTS

Alice's Operation Log			Bob's Operation Log			Clark's Operation Log		
Operation	logical vector	physical vector	Operation	logical vector	physical vector	Operation	logical vector	physical vector
$W(a)$	$<1,0,0>$	$<1,0,0>$	$W(c)$	$<0,1,0>$	$<0,1,0>$	$R(c)$	$<0,0,1>$	$<0,0,1>$
$W(b)$	$<3,0,0>$	$<5,0,0>$	$R(c)$	$<2,4,0>$	$<2,5,0>$	$R(d)$	$<0,0,2>$	$<0,0,4>$
$R(b)$	$<5,3,5>$	$<8,3,7>$	$W(d)$	$<2,5,0>$	$<2,6,0>$	$R(a)$	$<2,3,5>$	$<2,3,10>$

2 as follows: First, before constructing the graph, we move all writes that do not have any dictated reads. This is because only reads can reveal violations by their values. Second, we move redundant time edges. For two operations op_1 and op_2, a time edge is added from op_1 to op_2 only if $op_1 \rightarrow op_2$ and there is no op_3 that has the properties $op_1 \rightarrow op_3$ and $op_3 \rightarrow op_2$.

To provide the promised consistency, the data cloud should wait for a period of time to execute operations in the order of their logical vectors. For example, suppose that the logical vector of the latest write seen by the data cloud is $<0,1,0>$. When it receives a read from Alice with logical vector $<2,3,0>$, the data cloud guesses that there may be a write with logical vector $<0,2,0>$ coming from Bob. To ensure causal consistency, the data cloud will wait σ time to commit Alice's read, where σ is the maximal delay between servers in the data cloud. The maximal delay σ should also be written in the SLA. After waiting for $\sigma + \Delta$ time, where Δ is the maximal delay between the data cloud and the audit cloud, if the user still cannot get a response from the data cloud, or the response violates the promised consistency, he can claim that the data cloud violates the SLA.

C. Effectiveness

The effectiveness of the local consistency auditing algorithm is easy to prove. For monotonic-read consistency, a user is required to read either the same value or a newer value. Therefore, if the dictating write of a new read happens before the dictating write of the last read, we conclude that monotonic-read consistency is violated. For read-your-write consistency, the user is required to read his latest write. Therefore, if the dictating write of a new read happens before his last write, we conclude that read-your-write consistency is violated.

For causal consistency, we should prove that: (1) If the constructed graph is not a DAG, there must be a violation; (2) If the constructed graph is a DAG, there is no violation. It is easy to prove proposition (1). If a graph has a cycle, then there exists an operation that is committed before itself, which is impossible. We prove proposition (2) by contradiction. Assume that there is a violation when the graph is a DAG. A violation means that, given two writes $W(a)$ and $W(b)$ that have causal relationships $W(a) \rightarrow W(b)$, we have two reads $R(b) \rightarrow R(a)$. According to our construction, there must be a time edge from $W(a)$ to $W(b)$, a time edge from $R(b)$ to $R(a)$, a data edge from $W(a) \rightarrow R(a)$, and a data edge from $W(b) \rightarrow R(b)$. Therefore, there is a route $W(a)W(b)R(b)W(a)$, where the source is the dictating write $W(a)$ and the destination is the dictated read $R(a)$. Since there is a write $W(b)$ on the route, according to our rule, a causal edge from $W(b)$ to $W(a)$ will be added. This will cause a cycle, and thus contradicts our assumption.

D. Garbage Collection

In the auditing process, each user should keep all operations in his UOT. Without intervention, the size of the UOT would grow without bound. Furthermore, the communication cost for transferring the UOT to the auditor will be excessive. Therefore, we should provide a garbage collection mechanism which can delete unneeded records, while preserving the effectiveness of auditing.

In our garbage collection mechanism, each user can clear the UOT, keeping only his last read and last write, after each global consistency verification. This makes sure that a user's last write and last read will always exist in his UOT. In local consistency auditing, if the dictating write of a new read does not exist in the user's UOT and the dictating write is issued by the user, the user concludes that he has failed to read his last updates, and claims that read-your-write consistency is violated. If the dictating write of this read happens before the dictating write of his last read recorded in the UOT, the user concludes that he has read an old value, and claims that monotonic-read consistency is violated. If the dictating write of a new read does not exist in the user's UOT and the dictating write comes from other users, then a violation will be revealed by the auditor. In global consistency auditing, if there exists a read that does not have a dictating write, then the auditor concludes that the value of this read is too old, and claims that causal consistency is violated.

V. QUANTIFYING SEVERITY OF VIOLATIONS

As in [9], we provide two metrics to quantify the severity of violations for the CaaS model: commonality and staleness. Commonality quantifies how often the violations happen. Staleness quantifies how much older the value of a read is compared to that of the latest write. Staleness can be further classified into time-based staleness and operation-based staleness, where the former counts the passage of time, and the latter counts the number of intervening operations, between the read's dictating write and the latest write.

Commonality. For local consistency, commonality can be easily quantified by letting each user set a local counter and increasing the counter by one when a local consistency violation is revealed. During global auditing, each user sends the local counter as well as his UOT to the auditor, which will calculate the total number of local violations by summing all users' local counters. For global consistency, commonality can be quantified by counting the number of cycles in the constructed graph. This can be transformed into removing the minimum number of edges to make the graph acyclic. This measurement is coarse because an edge can be part of multiple cycles, but it is still informative. To illustrate, let us consider the cyclic directed graph shown in Fig. 5. After we remove

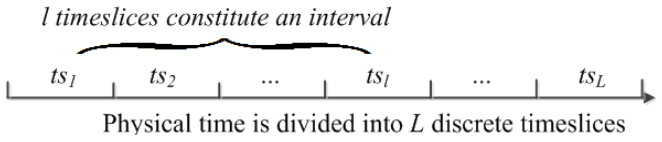

Fig. 6. Physical time is divided into timeslices.

one edge, $W(d)W(a)$, this graph becomes DAG. In reality, $W(a)$ happens before $W(b)$, thus Clark reads an old value a after reading d, and the commonality of violations is 1.

Staleness. Staleness is quantified in the same way in both local auditing and global auditing. Let $R(a)$ denote the user's current read, and let $W(b)$ denote the latest write. Given N users, for $R(a)$'s dictating write $W(a)$, the logical vector is $LV(W(a)) =< LC(1), \ldots, LC(N) >$, and the physical vector is $PV(W(a)) =< PC(1), \ldots, PC(N) >$; for the latest write $W(b)$, the logical vector is $LV(W(b)) =< LC(1)', \ldots, LC(N)' >$, and the physical vector is $PV(W(b)) =< PC(1)', \ldots, PC(N)' >$. We calculate $\sum_{i=1}^{N}(LC(i)' - LC(i))$ to denote operation-based staleness. This counter is coarse because the logical clock will increase by one whenever an event happens. When there are multiple concurrent latest writes, we use the maximal value as the operation-based staleness.

Since users in the audit cloud will exchange messages frequently, loose clock synchronization can be easily achieved by using the solution in [10]. Suppose that the maximal time difference between any two users is θ. If $W(a)$ is issued by user i and $W(b)$ is issued by user j, then time-based staleness is calculated with $|PC(j)' - PC(i)| + \theta$. If both writes are issued by the same user i, then time-based staleness is calculated with $|PC(i)' - PC(i)|$. When there are multiple concurrent latest writes, we use the maximal value as the time-based staleness.

For example, in Table I, a violation is revealed when user Clark reads data with value a from the data cloud. For $R(a)$'s dictating write $W(a)$, its logical vector is $< 1, 0, 0 >$, and its physical vector is $< 1, 0, 0 >$. The last write of Alice is $W(b)$ with logical vector $< 3, 0, 0 >$ and physical vector $< 5, 0, 0 >$; the last write of Bob is $W(d)$ with logical vector $< 2, 5, 0 >$ and physical vector $< 2, 6, 0 >$. Since $W(b)||W(d)$, we calculate the operation-based staleness with $\max(((3-1)+(0-0)+(0-0)), ((2-1)+(5-0)+(0-0))) = 6$. The time-based staleness is calculated as follows: $W(a)$ and $W(b)$ are issued by Alice, and thus the time is $|5-1| = 4$; $W(a)$ and $W(d)$ are issued from Alice and Bob, respectively, and thus the time is $|6-1| + \theta = 5 + \theta$. Then, time-based staleness is $\max(4, (5+\theta)) = 5 + \theta$.

VI. Heuristic Auditing Strategy

From the auditing process in the CaaS model, we observe that only reads can reveal violations by their values. Therefore, the basic idea of our heuristic auditing strategy (HAS) is to add appropriate reads for revealing as many violations as possible. We call these additional reads *auditing reads*.

As shown in Fig. 6, HAS divides physical time into L timeslices, where l timeslices constitute an interval. Each timeslice is associated with a state, which can be marked with either *normal* or *abnormal*. A normal state means that there is no consistency violation, and an abnormal state means that there is one violation in this timeslice.

HAS determines the number of auditing reads in the $(i+1)$-*th* interval, based on the number of abnormal states in the i-*th* interval. Let n_i denote the number of auditing reads in interval i. HAS determines n_{i+1}, which is the number of auditing reads in the next interval with Eq. 1:

$$\begin{cases} n_{i+1} = \min(l, k \times n_i), & n_i \geq \alpha \\ n_{i+1} = \max(1, \frac{1}{k} \times n_i), & n_i < \alpha \end{cases} \quad (1)$$

where k is a parameter that is used to adjust the value of n_{i+1}, l is the number of timeslices in an interval, and α is a threshold value that is used to determine whether the number of auditing reads in the next round should be increased by k times or be reduced to $1/k$, compared to the number of auditing reads in the current round.

Specifically, given a threshold value α, if a user issues n_i auditing reads and reveals more than α violations in interval i, in interval $i+1$, the user will issue $n_{i+1} = \min(l, k * n_i)$ auditing reads; that is, each timeslice will be issued, at most, one auditing read, and the maximal number of auditing reads will not exceed l. Otherwise, the user will issue $n_{i+1} = \max(1, \frac{1}{k} \times n_i)$ auditing reads, that is, each interval will be issued at least one auditing read. Since the number of auditing reads should be an integer, $\frac{1}{k} \times n_i$ is actually the abbreviation of $\lfloor \frac{1}{k} \times n_i \rfloor$.

Suppose that the SLA stipulates that the audit cloud can gain s (e.g., monetary compensation) from the data cloud if a consistency violation is detected, and that the audit cloud will be charged r for a read operation. If after executing n auditing reads, the users reveal v violations, then the earned profits P can be calculated by $P = s * v - r * n$.

Under the CaaS model, consistency becomes a part of the SLA, the users can obtain proportional compensation from the CSP, by revealing consistency violations and quantifying the severity of the violations. We believe that the CaaS model will help both the CSP and the users adopt consistency as an important aspect of cloud services offerings.

VII. Evaluation

In this section, we compare HAS with a random strategy, denoted as Random. To verify the effectiveness of HAS, we conduct experiments on both synthetic and real violation traces. Our experiments are conducted with MATLAB R2010a running on a local machine, with an Intel Core 2 Duo E8400 3.0 GHz CPU and 8 GB Linux RAM.

A. Synthetic Violation Traces

We summarize the parameters used in the synthetic violation traces in Table II. In the random strategy, we randomly choose $[1, l]$ auditing reads in each interval, where l is the length of an interval. To obtain the synthetic violation traces, physical time is divided into 2,000 timeslices. We assume that once a data cloud begins to violate the promised consistency, this violation will continue for several timeslices, rather than ending immediately. In the simulation, the duration of each violation d is set to 3-10 timeslices.

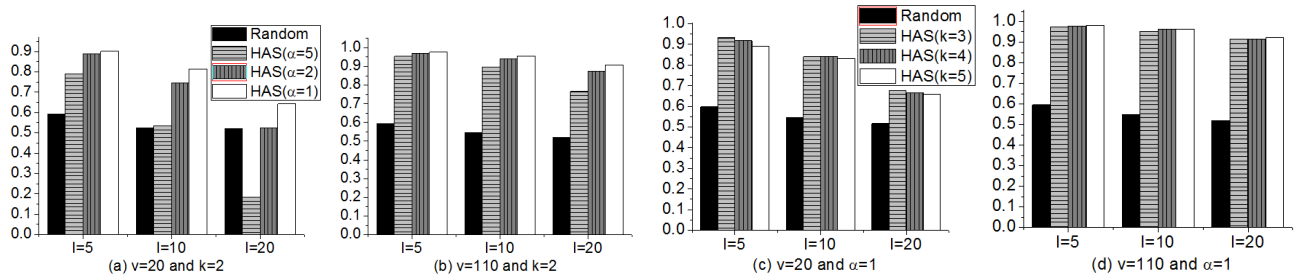

Fig. 7. Comparison of percentage of revealed violations. (a) and (b) have the same threshold values. (c) and (d) have the same adjusting values.

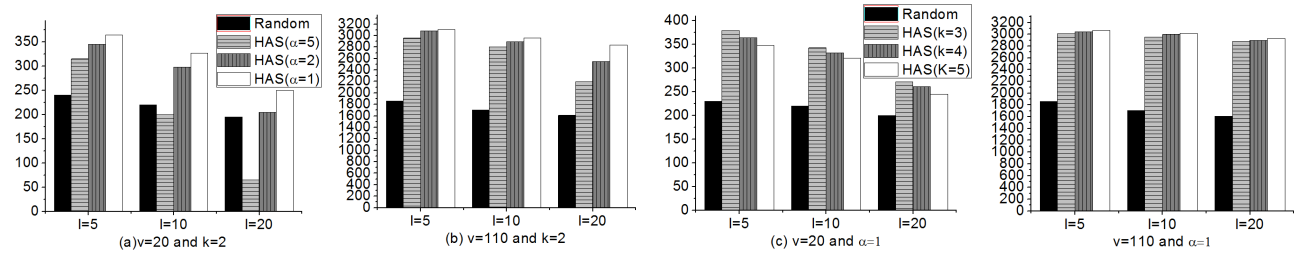

Fig. 8. Comparison of earned profits. (a) and (b) have the same threshold values. (c) and (d) have the same adjusting values.

TABLE II
PARAMETERS

Notation	Description	Value
L	Length of physical time	2,000 timeslices
l	Length of interval	5, 10, 20 timeslices
d	Duration of violation	3-10 timeslices
v	Number of violations	20, 110
k	Adjusting value	2, 3, 5
α	Threshold value	$1, 2, 5$
s	Gained money	\$5
r	Charged money	\$0.1

From Eq. 1, we know that three parameters will impact HAS: the length of an interval l, the threshold value α, and the adjusting parameter k. Therefore, our simulations are conducted with different parameter values. Specifically, the values of l are set to 5, 10, 20, respectively; the values of α are set to 1, 2, 5, respectively; and the values of k are set to 2, 3, 5, respectively. Furthermore, to show the impact of the violation frequency on HAS, the number of violations v is first set to 20 and then 110. When v is set to 20, the ratio of the number of abnormal timeslices to the length of physical time will be smaller than $1:10$. When v is set to 110, the ratio is about $1:2$.

To minimize the deviations, we run simulations $10,000$ times and calculate the average values. Fig. 7 shows the comparison results regarding the percentage of revealed violations. From Fig. 7, we know that HAS performs worse when either the threshold value α or the length of interval l increases, but performs better when the frequency of violation v increases. For example, when $l = 5$, HAS can reveal 90% of the violations under the setting $v = 20$ and $\alpha = 1$. As the length of an interval increases to 20, HAS only reveals 65% of the violations; when $\alpha = 1$, HAS can reveal 81% of the violations under the settings, $v = 20$ and $l = 10$. As the threshold value increases to 5, HAS only reveals 53% of the violations. Furthermore, HAS is affected by the adjusting value k. For example, under the settings, $v = 20$, $l = 5$ and $\alpha = 1$, while k changes from 2 to 5, the percentage of revealed violations in HAS is reduced from 90% to 82%.

Suppose that the audit cloud can gain \$5 from the data cloud once a consistency violation is detected, the audit cloud will be charged \$0.1 for an auditing read operation. Fig. 8 shows the comparison results of the earned profit P. From Fig. 8, we know that HAS usually earns a higher profit than Random. For example, when $v = 110$ and $k = 2$, HAS can generate about \$2,805, \$2,895, \$2,955, under the settings, of $\alpha = 5$, $\alpha = 2$, and $\alpha = 1$, respectively; however, Random only generates \$1,705. Furthermore, as the number of violations increases, HAS can generate a higher profit. For example, when $\alpha = 1$, $l = 5$, and $k = 2$, HAS generates \$365 under the settings, $v = 20$, and generates \$3,110 under the setting $v = 110$. Finally, HAS will generate higher earned profit as the parameters α and l decrease. As k increases, HAS will generate higher profit under the setting $v = 110$, but will generate lower profit under the setting $v = 20$.

Summary. HAS can detect almost all of the violations when the threshold value α and interval length l are chosen properly; Random can detect only about 60% of violations. Although HAS requires the auditing cloud to issue more auditing reads, the earned profit is still higher than Random. Specifically, as the parameters α and l decrease, HAS works better. However, as the number of violations v increases, the impact of parameters α and l on HAS decreases. Furthermore, when there are a lot of violations, the auditing cloud can use a large k value to earn a higher profit. Otherwise, a larger k value will generate a lower profit.

B. Real Violation Traces

To validate the effectiveness of HAS, we collect violation traces from two real clouds, TCloud[5] and Amazon EC2[6].

[5]https://sites.google.com/a/temple.edu/tcloud/home. This is the Temple Elastic HPC Cloud project site. The hardware is purchased under NSF grant CNS 0958854. This side documents the construction process of a private HPC cloud, lessons and experiences. This site is open to the public.

[6]http://aws.amazon.com/ec2/

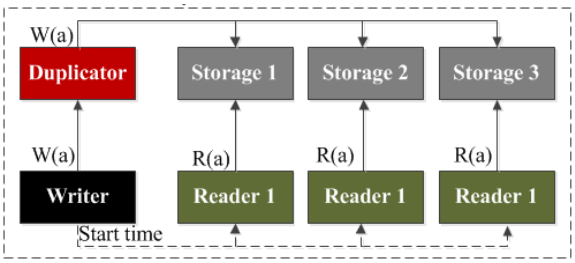

Fig. 9. Deployment in the TCloud and Amazon EC2.

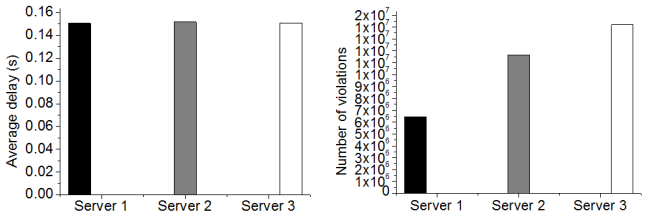

Fig. 10. Average delays and violations in TCloud.

The deployment in the TCloud and Amazon EC2 is shown in Fig. 9, with one *writer*, one *duplicator*, three *storages*, and three *readers*. The main difference on the deployments between TCloud and Amazon EC2 is that we initiate 8 instances in the TCloud, and 4 instances in Amazon EC2 (a writer and three readers are deployed in local machines). The writer continues to write unique data to the duplicator, which will propagate data to three storages in the order of receiving the data. Furthermore, the writer will send the time of completing each write as the *start time* to three readers, and each reader will record the time of reading new data as the *finish time*. The difference of the finish time and the start time is used as the *delay*. We use network time protocol (NTP) to synchronize time among all instances.

Parameter setting in TCloud. We subscribe $m1$.xlarge type instances that have 2 CPUs, 2GB RAM, and 20 GB of storage, running on a 64-bit Linux platform. The average bandwidth between the instances is 945.5 MB/s. The writer continues to write data for 1 day, and it is required to stop for 1 second, and then continues again, after completing 256 writes. The total number of writes is 22,118,400, and the average delay is shown in the left side of Fig. 10. We use the average delays as the threshold values to test violations as follows: For each reader, if a delay is larger than the threshold value, then the number of violations increases by one. The right side of Fig. 10 shows the total number of violations recorded by three readers. In the experiments, we let the number of time slices L be the same as the number of writes, i.e., 22,118,400. We set l, the length of intervals, to be 5, 10, 20, 50, and 100, respectively. Unlike the synthetic traces that set α with fixed values, we let the values of α depend on the values of l. Specifically, we set the values of α to $l/5$ and l, respectively. That is to say, when $l = 5$, the values of α are 1 and 5, when $l = 10$, the values of α are 2 and 10, and so on. Finally, the values of k are set to 2 and 5, respectively.

The comparisons of Random and HAS under different parameter settings are shown in Figs. 11 and 12, in each of which the right side is the comparison of HAS under

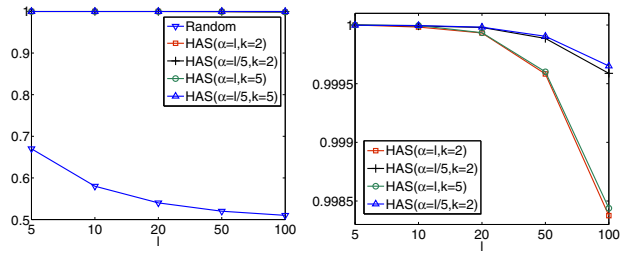

Fig. 11. Comparison of percentage of revealed violations in TCloud.

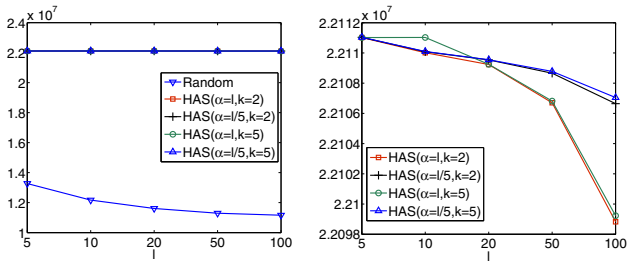

Fig. 12. Comparison of number of reads in TCloud.

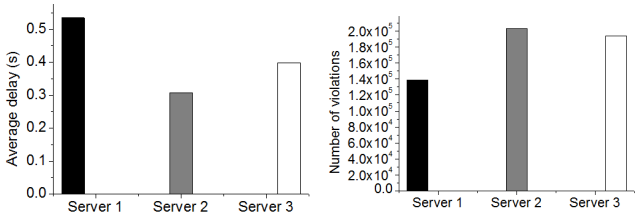

Fig. 13. Average delays and violations in Amazon EC2.

different parameter settings. From Fig. 11, we know that the percentage of revealed violations decreases as l increases, in terms of both HAS and Random. However, the change of l's value has less impact on HAS than Random. For example, while l increases from 5 to 100, the percentage of revealed violations, decreases from 99.99% to 99.97% in HAS with the setting of $\alpha = l/5$ and $k = 2$, and decreases from 67% to 50% in Random. Furthermore, HAS always reveals more violations than Random. For example, even when $l = 100$, HAS, which can reveal more than 99% of violations, still performs much better than Random, which can reveal only 50%. The right side of Fig. 11 shows the detailed comparisons of HAS under different values of α and k. We know that the percentage of revealed violations decreases as α increases or k decreases. However, these parameters have minor impacts on the percentage of revealed violations.

Parameter setting in Amazon EC2. We subscribe EC2 amzn-ami-2011.02.1.i386-ebs (ami-8c1fece5) AMI and a small type instance with the following specs: 32-bit platform, a single virtual core equivalent to 1 compute unit CPU, and 1.7 GB RAM. The average bandwidth from EC2 to the local machine is 33.43 MB/s, and from the local machine to EC2 is 42.98 MB/s. The writer continues to write data for 1 hour, and it is is required to stop for 1 second, and then continue again, after completing 256 writes. The total number of writes

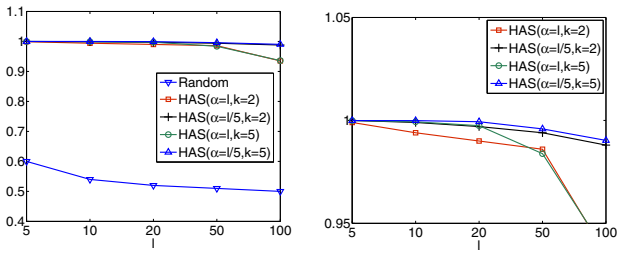

Fig. 14. Comparison of percentage of revealed violations in Amazon EC2.

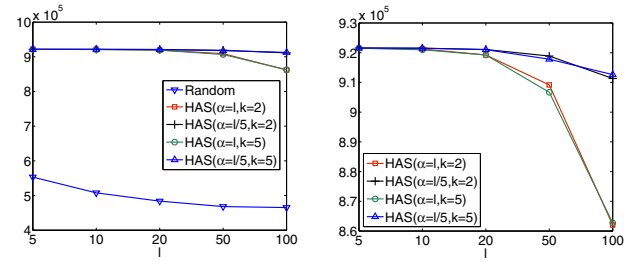

Fig. 15. Comparison of number of reads in Amazon EC2.

is 921,600, and the average delay is shown in the left side of Fig. 13. We use the average delays as the threshold values to test violations, and the total number of violations recorded by three readers are shown in right side of Fig. 13. As in the setting of TCloud, we let the number of timeslices L be the same as the number of writes, i.e., 921,600. We also set l, α and k with the same values as those in TCloud.

The comparisons of Random and HAS under different parameter settings are shown in Figs. 14 and 15, in each of which the right side is the comparison of HAS under different parameter settings. From Fig. 14, we know that the percentage of revealed violations decreases in terms of both HAS and Random, as l increases. However, even when $l = 100$, HAS still performs much better than Random. The right side of Fig. 14 shows the detailed comparisons of HAS under different values of α and k. We know the percentage of revealed violations decreases as α increases, e.g., this percentage decreases from 0.988 to 0.935, as α increases from $l/5$ to l, when $l = 100$, and $k = 2$, but k has minor impacts on the percentage of revealed violations, e.g., as k increases from 2 to 5, this percentage only increases from 0.988 to 0.990 when $l = 100$ and $\alpha = l/5$.

Summary. The performance of Amazon EC2 is more stable than that of TCloud. The violations occur less frequently in Amazon EC2. The main reason for this may be that all of the 8 instances run in TCloud, but only 4 instances run in Amazon EC2. Furthermore, the longer running time in TCloud may be another factor. The experimental results in the real cloud traces are similar to those in the synthetic traces, e.g., HAS usually works better than Random, and as the parameters α and l decrease, HAS can detect more violations. However, since there are a lot of violations in both TCloud and Amazon EC2, the changes of the parameters α, l, and k make minor differences. For example, the percentage of revealed violations in TCloud decreases from 0.999 to 0.998, as α increases from $l/5$ to l, when $l = 100$, and $k = 2$; this percentage decreases from 0.996 to 0.995, as k decreases from 5 to 2, when $l = 100$ and $\alpha = l/5$. Note that HAS works better as there are more violations, e.g., HAS can detect almost all of the violations under different parameter settings, when there are a lot of violations in real cloud traces. Therefore, the feasibility of HAS is verified.

VIII. Discussion

In this section, we will discuss some additional issues about CaaS in terms of the election of an auditor and other consistency models.

A. Election of An Auditor

In section III, an auditor is simply elected from the auditor cloud in turn, where each user becomes the auditor with the same probability. However, different users have different levels of ability in terms of available bandwidth, CPU, and Memory of clients. The users with a higher ability should have a higher probability of being selected as auditor. In this section, we provide a more comprehensive solution to elect an auditor as follows: We construct an ID ring for a group of users, where each node is associated with a node ID, and each user is denoted by a set of nodes in the ring. Suppose the number of nodes in the ring is n. To elect an auditor, we can randomly generate a number r, and let the user who is denoted by the node with an ID of $(r \mod n)$ in the ring to be the auditor.

Note that the selection of each user does not have to be uniform. The number of nodes associated with a user can be determined by his abilities, e.g., the capability of his client, his trusted rank, and so on. In this way, the probability of a user with a higher ability of being chosen as the auditor becomes higher. For example, given 3 users and 6 nodes, user Alice is denoted by 3 nodes, user Bob is denoted by 2 nodes, and user Clark is denoted by 1 node. Therefore, the probability of Alice being the auditor is 50%, for Bob it is 33%, and for Clark it is 17%.

B. Other Consistency Models

In local auditing, we only consider two kinds of consistencies, i.e., monotonic-read consistency and read-your-write consistency. Now, we discuss other consistency models such as read-after-write consistency and monotonic-write consistency models. Read-after-write consistency requires that all clients immediately see new data. With read-after-write consistency, a newly created object, file, or table row will immediately be visible, without any delays. Therefore, we can build distributed systems with less latency. Today, Amazon S3 provides read-after-write consistency in the EU and US-west regions. So far, it is hard to achieve read-after-write consistency in a world-wide scale.

Monotonic-write consistency requires that a write on a copy of data item x is performed only if that copy has been updated by any preceding write operations that occurred in other copies. However, monotonic-write is not always necessary for all applications. For example, the value of x is first set to 4 and, later on, is changed to 7. The value 4 that has been overwritten isn't really important.

IX. Conclusion

In this paper, we presented a consistency as a service (CaaS) model and a two-level auditing structure to help users verify whether the cloud service provider (CSP) is providing the promised consistency, and to quantify the severity of the violations, if any. With the CaaS model, the users can assess the quality of cloud services and choose a right CSP among various candidates, e.g, the least expensive one that still provides adequate consistency for the users' applications. For our future work, we will conduct a thorough theoretical study of consistency models in cloud computing.

Acknowledgments

This research was supported in part by NSF grants ECCS 1231461, ECCS 1128209, CNS 1138963, CNS 1065444, and CCF 1028167; and NSFC grants 61272151 and 61073037, ISTCP grant 2013DFB10070, the China Hunan Provincial Science & Technology Program under Grant Number 2012GK4106, and the "Mobile Health" Ministry of Education-China Mobile Joint Laboratory (MOE-DST No. [2012]311).

References

[1] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, 2010.

[2] P. Mell and T. Grance, "The NIST definition of cloud computing (draft)," NIST Special Publication 800-145 (Draft), 2011.

[3] E. Brewer, "Towards robust distributed systems," in *Proc. 2000 ACM PODC*.

[4] ——, "Pushing the CAP: strategies for consistency and availability," *Computer*, vol. 45, no. 2, 2012.

[5] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto, "Causal memory: definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, 1995.

[6] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS," in *Proc. 2011 ACM SOSP*.

[7] E. Anderson, X. Li, M. Shah, J. Tucek, and J. Wylie, "What consistency does your key-value store actually provide," in *Proc. 2010 USENIX HotDep*.

[8] C. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," in *Proc. 1988 ACSC*.

[9] W. Golab, X. Li, and M. Shah, "Analyzing consistency properties for fun and profit," in *Proc. 2011 ACM PODC*.

[10] A. Tanenbaum and M. Van Steen, *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, 2002.

[11] W. Vogels, "Data access patterns in the Amazon.com technology platform," in *Proc. 2007 VLDB*.

[12] ——, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, 2009.

[13] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska, "Building a database on S3," in *Proc. 2008 ACM SIGMOD*.

[14] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann, "Consistency rationing in the cloud: pay only when it matters," in *Proc. 2009 VLDB*.

[15] S. Esteves, J. Silva, and L. Veiga, "Quality-of-service for consistency of data geo-replication in cloud computing," *Euro-Par 2012 Parallel Processing*, vol. 7484, 2012.

[16] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data consistency properties and the trade-offs in commercial cloud storages: the consumers' perspective," in *Proc. 2011 CIDR*.

[17] D. Bermbach and S. Tai, "Eventual consistency: how soon is eventual?" in *Proc. 2011 MW4SOC*.

[18] M. Rahman, W. Golab, A. AuYoung, K. Keeton, and J. Wylie, "Toward a principled framework for benchmarking consistency," in *Proc. 2012 Workshop on HotDep*.

[19] D. Kossmann, T. Kraska, and S. Loesing, "An evaluation of alternative architectures for transaction processing in the cloud," in *Proc. 2010 ACM SIGMOD*.

[20] L. Lamport, "On interprocess communication," *Distributed Computing*, vol. 1, no. 2, 1986.

[21] A. Aiyer, L. Alvisi, and R. Bazzi, "On the availability of non-strict quorum systems," *Distributed Computing*, vol. 3724, 2005.

[22] J. Misra, "Axioms for memory access in asynchronous hardware systems," *ACM Trans. Programming Languages and Systems*, vol. 8, no. 1, 1986.

[23] P. Gibbons and E. Korach, "Testing shared memories," *SIAM J. Computing*, vol. 26, no. 4, 1997.

[24] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proc. 2007 ACM SOSP*.

[25] T. Gormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 1990.

PLACE PHOTO HERE

Qin Liu received B.Sc. in Computer Science in 2004 from Hunan Normal University, China, and received M.Sc. in Computer Science in 2007 and Ph.D. in Computer Science in 2012 from Central South University, China. She has been a Visiting Student at Temple University, USA. Her research interests include security and privacy issues in cloud computing.

PLACE PHOTO HERE

Guojun Wang received B.Sc. in Geophysics, M.Sc. in Computer Science, and Ph.D. in Computer Science, from Central South University, China. He is now Chair and Professor of Department of Computer Science at Central South University. He is also Director of Trusted Computing Institute of the University. He has been an Adjunct Professor at Temple University, USA; a Visiting Scholar at Florida Atlantic University, USA; a Visiting Researcher at the University of Aizu, Japan; and a Research Fellow at the Hong Kong Polytechnic University. His research interests include network and information security, Internet of things, and cloud computing. He is a senior member of CCF, and a member of IEEE, ACM, and IEICE.

PLACE PHOTO HERE

Jie Wu is the chair and a Laura H. Carnell Professor in the Department of Computer and Information Sciences at Temple University. Prior to joining Temple University, he was a program director at the National Science Foundation and Distinguished Professor at Florida Atlantic University. His current research interests include mobile computing and wireless networks, routing protocols, cloud and green computing, network trust and security, and social network applications. Dr. Wu regularly published in scholarly journals, conference proceedings, and books. He serves on several editorial boards, including IEEE Transactions on Computers, IEEE Transactions on Service Computing, and the *Journal of Parallel and Distributed Computing*. Dr. Wu was general co-chair/chair for IEEE MASS 2006 and IEEE IPDPS 2008 and program co-chair for IEEE INFOCOM 2011. Currently, he is serving as general chair for IEEE ICDCS 2013 and ACM MobiHoc 2014, and program chair for CCF CNCC 2013. He was an IEEE Computer Society Distinguished Visitor, ACM Distinguished Speaker, and chair for the IEEE Technical Committee on Distributed Processing (TCDP). Dr. Wu is a CCF Distinguished Speaker and a Fellow of the IEEE. He is the recipient of the 2011 China Computer Federation (CCF) Overseas Outstanding Achievement Award.