Volume 5, Issue 8, August 2015 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Provision of Consistency as a Service in Cloud through Cloud Auditing

Sowmya C*, Shashirekha H
Dept. of CS & E, VTU RC,
Mysuru, India

***Abstract*— Cloud Computing is popular due to its commercial importance. There is no time limit to access the data and data is available on anytime and anywhere. By using replication technique achieving strong consistency becomes difficult. The proposed system presents Consistency model. It has a data cloud maintained by cloud service provider, and multiple audit cloud formed by group of users. Audit cloud performs verification of consistency provided by the cloud service provider. The proposed system adopts two level auditing architecture. The consistency algorithms are designed to measure the violations**.

***Keywords*— Consistency as a service (CaaS), two level auditing strategy, local consistency, global consistency.**

I. INTRODUCTION

By the trend of everything as a service (XaaS) model data storage, virtual infrastructure and virtual platforms software and applications are provided and consumed as service in cloud. Many CSPs like Amazon DB, Microsoft Azure etc charge for the services on the utility basis, e.g., per gigabyte per month. It is too costly to achieve strong consistency using cloud due to replication on world wide scale. Using cloud storage devices users can access data stored in cloud anywhere anytime using any device, without carrying large amount of capital investment when deploying the underlying hardware infrastructure. Many CSPs (e.g., Amazon s3) only eventual consistency which is a weak consistency for high performance and availability where user reads the stale data. Updates to the names will not be visible immediately but clients are ensured to see them eventually, e.g., is DNS.

Eventual consistency is not suitable for all applications. For interactive applications casual consistency is required. Consider the following scenario shown in fig1.

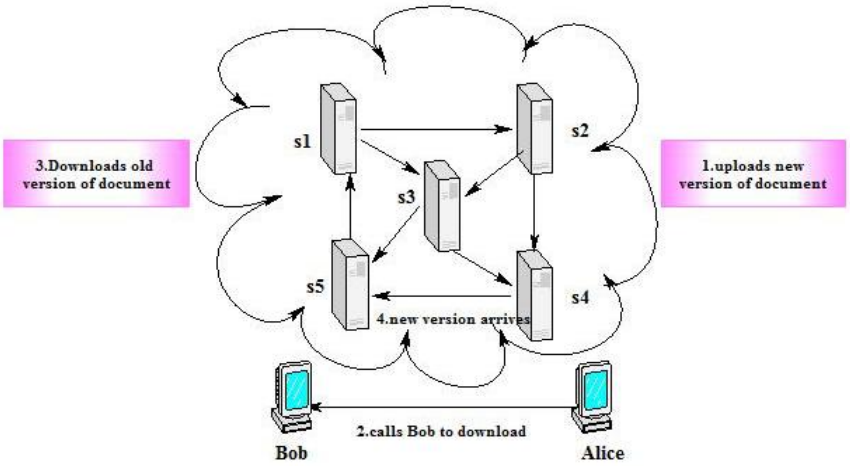

Fig1 An Application that requires casual consistency

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, . . ., *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 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 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.

II. SOLUTION STRATEGY

Consistency as a Service model

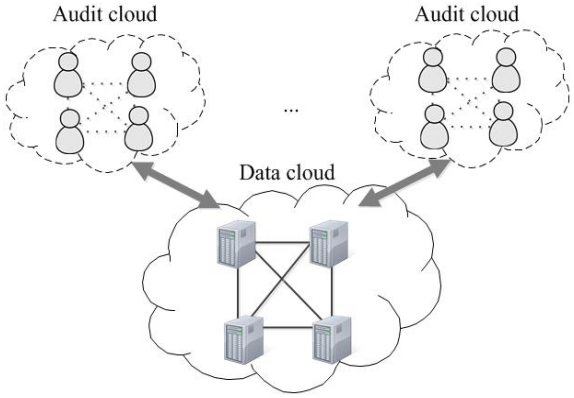

Fig 2 CaaS model

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. 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.

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.

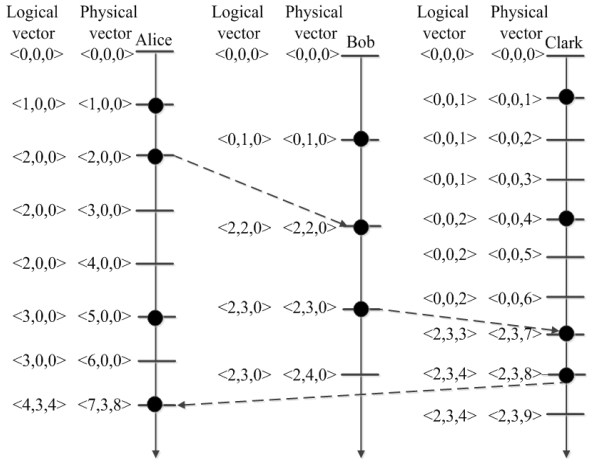

Fig. 3. The update process of logical vector and physical vector. A black solid circle denotes an event (read/write/send message/receive message), and the arrows from top to bottom denote the increase of physical time.

Each user will maintain a logical vector and a physical vector to track the logical and physical time when an operation happens, respectively. 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 IDi where $1 \leq i \leq N$, his logical vector is $< LC1, LC2, \ldots, LCN >$, where LCi is his logical clock, and LCj is the latest logical clock of user j to his best knowledge; his physical vector is $< PC1, PC2, \ldots, PCN >$, where PCi is his physical clock, and PCj is the latest physical clock of user j, to the best of his knowledge.

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).

Two Level Auditing Strategies

There are three types of consistencies

- Monotonic Read Consistency
- Read your Write Consistency
- Casual Consistency

At the first level these two consistencies are verified

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.

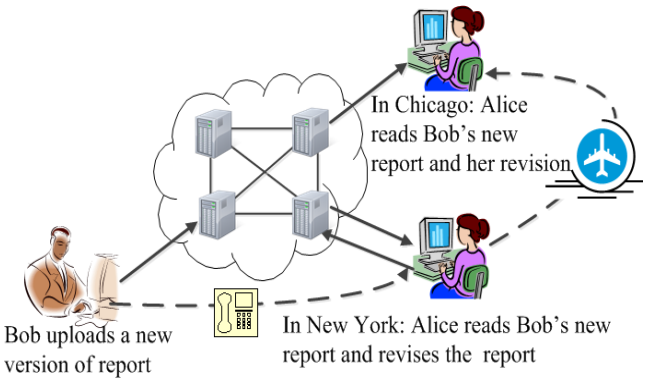

Fig4.Application that has different consistency requirements

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.

III. METHODOLOGY

We use two algorithms namely

- Local Consistency Auditing
- Global Consistency Auditing

Algorithm 1

Local consistency auditing

Initial UOT with $\emptyset$

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

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.

Algorithm 2

Global consistency auditing

Each operation in the global trace is denoted by avertex
for any two operations *op*1 and *op*2 **do**
if *op*1 → *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

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.

Verification of Consistency Property

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 → *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 1as 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*) → *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. 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 → *op*2 and there is no *op*3 that has the properties *op*1 → *op*3 and *op*3 → *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.

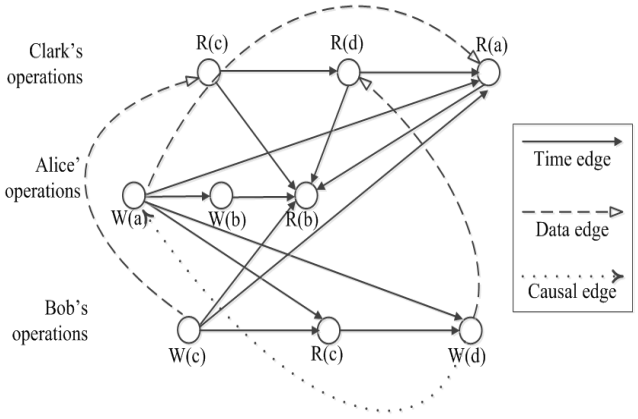

Fig.5. Sample graph constructed with Alg. 2.

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.

Table 1
Sample UOT's

Alice operation log

operation	logical vector	phisical vector
W(a)	<1,0,0>	<1,0,0>
W(b)	<3,0,0>	<5,0,0>
R(d)	<5,3,5>	<8,3,7>

Bobs operation log

operation	logical vector	phisical vector
W(c)	<0,1,0>	<0,1,0>
R(c)	<2,4,0>	<2,5,0>
W(d)	<2,5,0>	<2,6,0>

Clarks operation log

operation	logical vector	phisical vector
R(c)	<0,0,1>	<0,0,1>
R(d)	<0,0,2>	<0,0,4>
R(a)	<2,3,5>	<2,3,10>

IV. CONCLUSIONS

The proposed system states that the users are required to select the proper cloud service providers among others to use the cloud services by considering the level of consistency provided by the service providers. For further enhancement a detailed study of consistency models is performed.

REFERENCES

[1] Qin Liu, Guojun Wang, IEEE, Member, and Jie Wu, IEEE Fellow "Consistency as a service: Auditing cloud consistency", IEEE Vol 11 No.1, July 2014.

[2] A Tanenbaum And M. Van Steen, Distributed Systems: Principles and Paradigms. Prentice Hall PTR, 2002.
[3] W. Vogel's, ―Data access patterns in the Amazon.com technology platform,‖ in Proc. 2007 VLDB.
[4] E. Brewer, ―Towards robust distributed systems,‖ in Proc. 2000 ACM PODC.
[5] Pushing the CAP: strategies for consistency and availability,‖ Computer, vol. 45, no.2, 2012
[6] E. Anderson, X. Li, M. Shah, J. Tucek, And J. Wylie, ―What Consistency Does Your Key-Value Store Actually Provide,‖ In Proc. 2010 Use nix Hot Dep.
[7] W. Golab, X. Li, And M. Shah, ―Analyzing Consistency Properties For Fun And Profit,‖ In Proc. 2011 Acm Podc.
[8] 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.
[9] P. Mell and T. Grance, ―The NIST definition of cloud computing (draft),‖ NIST Special Publication 800-145 (Draft), 2011.
[10] Eventually consistent,‖ *Commun. ACM*, vol. 52, no. 1, 2009.
[11] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska, ―Building a database on S3,‖ in *Proc. 2008 ACM SIGMOD*.

