A Case for the Accountable Cloud

Andreas Haeberlen
Max Planck Institute for Software Systems (MPI-SWS)

Abstract

For many companies, clouds are becoming an interesting alternative to a dedicated IT infrastructure. However, cloud computing also carries certain risks for both the customer and the cloud provider. The customer places his computation and data on machines he cannot directly control; the provider agrees to run a service whose details he does not know. If something goes wrong – for example, data leaks to a competitor, or the computation returns incorrect results – it can be difficult for customer and provider to determine which of them has caused the problem, and, in the absence of solid evidence, it is nearly impossible for them to hold each other responsible for the problem if a dispute arises.

In this paper, we propose that the cloud should be made accountable to both the customer and the provider. Both parties should be able to check whether the cloud is running the service as agreed. If a problem appears, they should be able to determine which of them is responsible, and to prove the presence of the problem to a third party, such as an arbitrator or a judge. We outline the technical requirements for an accountable cloud, and we describe several challenges that are not yet met by current accountability techniques.

1 Introduction

Cloud computing is becoming increasingly popular. Among other benefits, it offers customers a way to obtain computation and storage resources 'on demand'. Rather than owning (and maintaining) a large and expensive IT infrastructure, customers can now rent the necessary resources as soon as, and as long as, they need them. Thus, customers can not only avoid a potentially large up-front investment (which is particularly attractive for small companies and startups), they may also be able to reduce their costs through economies of scale and by paying only for the resources they actually use.

However, from the customer's perspective, using a cloud is also somewhat risky because he must to a large extent relinquish control over his computation and data. In the conventional model, where the computation runs on a server farm on the customer's premises, the customer has physical access to the machines, he can directly observe their status, and he can have them managed by people he trusts. In the new model, where the computation runs on virtual machines in a cloud, the customer can do none of these things. Management of the physical machines is delegated to the cloud provider; the customer retains some control over only the virtual machines, which he can manage remotely over a network connection.

The loss of control is problematic when something goes wrong. To illustrate this point, here is just a small selection of potential problems:

- The machines in the cloud can be misconfigured or defective and can consequently corrupt the customer's data or cause his computation to return incorrect results;
- The cloud provider can accidentally allocate insufficient resources to the customer, which can degrade the performance of the customer's services and cause him to miss his SLAs;
- An attacker can exploit a bug in the customer's software to steal valuable data, or to take over the customer's machines for spamming or DoS attacks;
- The customer may not have access to his data because the cloud loses it, or simply because the data is unavailable at an inconvenient time.

Some of these problems, such as the inadequate allocation of resources, are cloud-specific and could not occur on a dedicated platform, whereas others, such as data loss or broken hardware, are old and venerable systems problems whose severity is merely increased by the use of a cloud. For example, machines can become defective whether they are owned by the customer or by the cloud, but in the former case the customer can mitigate the risk by performing regular upgrades and by hiring capable

system administrators. If the customer uses a cloud, he must trust the cloud provider to diligently perform these tasks.

Unfortunately, since the management responsibilities are divided between customer and provider, neither of them is in a good position to address these problems. Even detecting the presence of a problem can be surprisingly difficult: on the one hand, the provider does not know what to look for, since he does not know what the computation is supposed to do; on the other hand, the customer can only access the cloud machines remotely, so he has only very limited information. Moreover, when a problem is detected, the customer and the provider face the potentially difficult task of deciding which of them is responsible for it – it is quite natural for the provider to initially suspect a problem with the customer's software, and vice versa. And if such a dispute cannot be resolved amicably, it is nearly impossible for either of them to convince a third party (such as an arbitrator or a judge) that the other is responsible.

The absence of reliable fault detection and attribution may not only discourage potential cloud customers, it also complicates (or even rules out) certain applications that require compliance with laws or regulations, such as the Health Insurance Portability and Accountability Act (HIPAA), which strictly regulates the use and disclosure of protected health information (PHI). According to one of Amazon's own case studies, one of their customers had to modify their software architecture to remove the PHI before their data was uploaded to the cloud; the PHI was then kept locally and reconnected with the processed data upon its return [2].

We propose to use *accountability* [7, 13] to address these challenges. In a nutshell, we say that a distributed system is accountable if a) faults can be reliably detected, and b) each fault can be undeniably linked to at least one faulty node. More specifically, we consider systems that have the following features:

- **Identities:** Each action (such as the transmission of a message) is undeniably linked to the node that performed it;
- **Secure record:** The system maintains a record of past actions such that nodes cannot secretly omit, falsify, or tamper with its entries;
- **Auditing:** The record can be inspected for signs of faults; and
- **Evidence:** When an auditor detects a fault, it can obtain evidence of the fault that can be verified independently by a third party.

Accountability appears to be a promising approach to the problems we outlined above. Customers of an accountable cloud would be able to check whether the cloud is performing as agreed. If a problem occurs, the customer and the provider could use the evidence to decide who is responsible, and, if a dispute arises, they could present the evidence to a third party, such as an arbitrator or a judge.

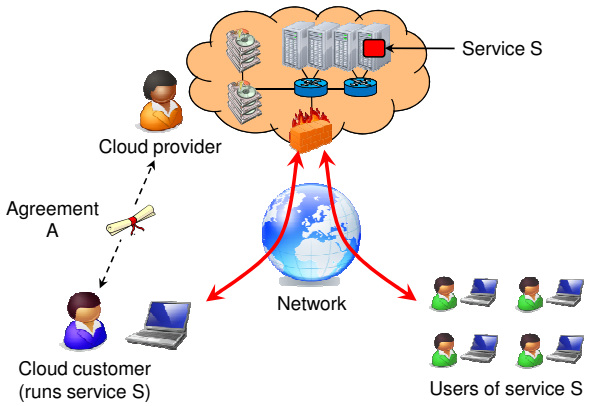

Figure 1: Cloud computing scenario. The customer runs a service on the cloud, but he has no physical control over the cloud machines, and he cannot directly observe their state.

However, existing accountability techniques fall short of the requirements for cloud computing in several ways. Since clouds are general-purpose platforms, the provider should be able to offer accountability for any service his customers may choose to run on it; this rules out application-specific techniques like CATS [14] or Repeat and Compare [10]. The application-independent technique in PeerReview [7], on the other hand, requires software modifications and assumes that the behavior of the software is deterministic, neither of which seems realistic in a cloud computing scenario. Finally, even if these limitations are overcome, the above techniques can only detect violations of a single property (correctness of execution); they were not designed to check other properties of interest in the cloud, such as conformance to SLAs, protection of confidential data, data durability, service availability, and so on.

This paper is intended as a call for action; its goal is to motivate further research on accountable clouds. At this time, we cannot present (nor do we claim to possess) a technical solution that would address all the challenges we outlined above. We do, however, define requirements for an accountability service, and we sketch a set of techniques that could form the basis of an accountable cloud.

2 The accountable cloud

In this section, we begin by discussing tradeoffs and challenges related to building accountable clouds, and we suggest a basic primitive called AUDIT that an accountable cloud could provide.

2.1 Problem statement

Figure 1 illustrates the scenario we are concerned with in this paper. A customer is interested in running a service S on the cloud, which (optionally) can be accessed by a group of external users. For this purpose, the customer rents a number of (physical or virtual) machines from a cloud provider.[1] Significantly, the customer does not have any physical control over the cloud machines, and he cannot directly observe their status.

The customer and the provider also enter into an agreement A that describes how the cloud is going to run service S. Typically, A specifies at least that the cloud machines will faithfully execute the software provided by the customer. However, A can also specify other properties, such as an SLA, an availability goal, or a promise that the data used by S will not be revealed to a third party.

Our goal is to implement a primitive called AUDIT(A, S, t_1, t_2) on the cloud that can be invoked by the customer to determine whether the cloud has fulfilled agreement A for service S during the interval $[t_1 \dots t_2]$. AUDIT returns either OK, to indicate that S has conformed to A in the specified interval, or some evidence that S has failed to conform to A.

2.2 What is the proper fault model?

We argue that an accountable cloud should be able to provide accountability even in the presence of Byzantine behavior [8], that is, even if the customer or the provider are outright malicious, and even if they collude with some or all of the users. At first glance, this requirement may seem overly pessimistic. After all, current cloud platforms are provided by reputable companies such as Amazon, Microsoft, or Google, and these companies are unlikely to intentionally commit fraud or act maliciously against their customers.

We do not deny that cloud providers are unlikely to be malicious. However, there are several other reasons why the Byzantine fault model is appropriate. First, intentional misbehavior is not the only source of Byzantine faults; hacker attacks, software bugs, and manipulations by disgruntled employees can cause similar effects, and their detection is no less important than that of malicious behavior. Second, for a general-purpose system like the cloud that is used in so many different ways, it seems difficult to come up with a more restricted fault model; any assumption about the nature of potential problems is likely to be violated.

Finally, consider the problem from the perspective of the cloud provider. Malicious behavior by customers is certainly a possibility; for example, criminals could try to extort money from the provider by threatening to tarnish his reputation, or competitors could try to frame him by generating lots of complaints. Hence, it is likely that most providers would refuse to offer accountability if there was even the slightest chance that a correctly operating cloud could be made to appear faulty. To get accountability adopted, we need excellent arguments – ideally, provable guarantees – to show that false positives are impossible, no matter what customers and users do.

[1] In this paper, we focus on virtual machine rental as an example of a general-purpose cloud service. However, we believe that other cloud services could benefit from accountability as well.

2.3 What are the provider's incentives?

From the customer's perspective, there are clear advantages to using an accountable cloud: he can detect if the cloud does not run the service as agreed, and he can hold the cloud provider responsible. From the cloud provider's perspective, however, accountability may appear more as a potential source of problems: it can make the cloud 'look bad' by revealing problems that might not otherwise have been noticed, and it shifts some power to the customer by providing him with evidence of faults. Why should the cloud provider agree to be accountable?

One obvious reason is that accountability is likely to be attractive to prospective customers. However, there are also more direct benefits to the cloud provider: he can use accountability to proactively detect and diagnose problems, and he can more easily handle customer complaints. Currently it is difficult for customers to distinguish between problems caused by the cloud, and problems they cause themselves; as a result, providers no doubt receive many complaints for which they are not responsible. If such a complaint arises with an accountable cloud, the customer and the provider can simply perform an audit to determine who is responsible.

2.4 Is privacy an issue?

In some scenarios, there is a tension between privacy and accountability, since the latter produces a detailed record of a machine's actions that can then be inspected by a third party. However, it is important to consider *what* is being recorded, and *who* the record is made available to. An accountable cloud could keep separate logs for each of its customers, and it could make each log available only to the customer who owns it. Thus, customer A would not learn anything about the actions of customer B (or even that other customers exist), and users would not learn anything about the actions of either customer because they would not be allowed to audit the cloud at all.

The question remains whether accountability would compromise the cloud provider's privacy towards its customers. At first glance, the answer is obvious, since the customers are paying the cloud provider and therefore

have every right to learn what is being done on their behalf. However, recall that AUDIT also returns evidence of faults. If the evidence showed which component (router, storage system, firewall, or server) of the cloud had caused the fault, the customer might be able to make inferences about the internal structure of the cloud. On the other hand, this information is obviously valuable to the provider, who is responsible for diagnosing and resolving the problem. To get around this, the cloud can return evidence at different levels of detail, depending on who invokes AUDIT.

2.5 Is fault tolerance an alternative?

As mentioned earlier, accountability can only detect and report faults, but not mask their symptoms. Thus, if a fault occurs in the cloud, it is possible that the customer or some of the users will be affected by it. A natural question to ask is why the customer should settle for such a seemingly weak guarantee – after all, a number of fault tolerance techniques are available, so shouldn't it be possible to mask faults?

The problem is that most fault tolerance techniques require strong assumptions about the environment, such as failure independence between components, or an upper bound on the number of components that can be faulty at any given time. There are ways to ensure that these assumptions hold, such as the use of multiple software versions or the physical separation of replicas. However, the customer cannot be sure whether the cloud provider has taken any of these steps. For example, two apparently independent virtual machines may be assigned to servers that are connected to the same router, or the servers may be managed by the same administrator.

Of course, we expect that most cloud providers will do their best to provide good service, and to mask as many faults as they can. However, we argue that it is *still* useful to have accountability, so that 1) the customer can check whether the cloud is really as reliable as promised, and 2) the customer and the provider together can detect and resolve any problems that the cloud fails to mask.

2.6 Proposed guarantees

If accountability is to be adopted, it must strike a balance between the requirements of the customer and those of the provider – that is, it must avoid both false negatives (the customer's concern) and false positives (the provider's concern), and it must provide them with evidence to resolve disagreements, possibly with the help of a third party. To achieve this, we propose that AUDIT should have at least the following three properties:

- **Completeness:** If the agreement is violated, AUDIT will report this eventually, and it will produce evidence of the violation;
- **Accuracy:** If the agreement is *not* violated, AUDIT will *not* report a violation; and
- **Verifiability:** Any evidence of an alleged violation can be checked independently by a third party, even if the third party trusts neither the customer nor the provider.

Note that these guarantees are not intended to be final, but rather as a basis for further discussion. There is no single 'correct' way to define accountability, and different variants may be appropriate for different applications.

2.7 Can these guarantees be relaxed?

Our proposed guarantees are very strong, and they rely on very few assumptions; for example, they do *not* require that providers and customers trust each other, or that the cloud will only be affected by faults of a particular type. Having rock-solid guarantees is certainly reassuring, particularly in a technique like accountability that is meant to be used when things are already going wrong. However, this strength is going to come at a price. Can we relax some of these guarantees to get different 'shades' of accountability?

Not all accountability guarantees can be relaxed safely. We envision accountability not merely as a tool for administrators, but as a technique for enforcing real-world contracts between companies and organizations. In this context, the detection of a fault can have serious legal and financial consequences for the responsible party. Hence, the accuracy guarantee is absolutely essential; an accountable party must not be blamed for faults caused by others, or for faults that did not actually occur. If accuracy were relaxed, we doubt that cloud providers would adopt accountability.

However, this concern does not necessarily apply for the other guarantees. For example, completeness could be made probabilistic; that is, the cloud could detect each instance of a fault only with high probability. This should be safe as long as neither the customer nor the provider can influence the detection process. Alternatively, some of the assumptions could be strengthened. For example, if there is a third party that is trusted by both the provider and her customers, auditing could be delegated to that party to reduce overhead. This form of auditing is common in the financial world, where direct audits by customers are impractical for various reasons.

3 Building blocks for an accountable cloud

Next, we describe a set of building blocks that could form the basis of an accountable cloud. For some of the building blocks, we can rely on existing work; others have yet to be developed, and we merely sketch them here.

3.1 Tamper-evident logs

To implement AUDIT, we need to know the past actions of the various entities in the system (customer, provider, users, cloud machines). For this purpose, we can use a *tamper-evident log*, e.g., the log from PeerReview [7]. Each node maintains a log in which it records all of its inputs and outputs, including any messages it sends or receives, and it allows certain other nodes to audit this log. The log is structured in such a way that auditors can detect if any entries have been omitted, modified, or otherwise tampered with. If the auditor detects tampering, he also obtains evidence that can be verified independently by a third party.

Tamper-evident logs can provide a solid basis for accountable clouds. If the customer is able to audit the logs of the cloud machines he is renting, he can be sure that he either obtains a correct record of each machine's past actions (which he can then inspect for signs of faults) or evidence that some machines have not been keeping their logs correctly and are therefore faulty. Tamper-evident logs also offer strong, provable guarantees; in particular, it is impossible to obtain valid evidence against a correct node. As mentioned in Section 2.2, such a guarantee can help to dispel some of the provider's concerns about being blamed for non-existent faults.

3.2 Virtualization-based replay

How can the customer recognize faults in the log? If the software is deterministic, he can simply replay the inputs in the log to a local instance of the software he has installed on the cloud, and compare the outputs to the outputs in the log. Incorrect state transitions cause a discrepancy and can thus be detected [7]. However, we cannot assume that the customer's software is necessarily deterministic, and it is not always feasible to make the modifications necessary for logging and replay (e.g., if the source code is not available). Fortunately, we can achieve a similar effect in another way (following [6]): during the original execution, we can run the unmodified software in a virtual machine, and we can record all nondeterministic inputs or events, such as interrupts or device I/O, that occur in the virtual machine. During an audit, we can then reproduce this execution by initializing another virtual machine with the same image, and by injecting the recorded inputs or events at the same points during the execution.

Existing cloud platforms like EC2 are already based on virtual machines, and it should not be difficult to add the functionality for logging and replay to their VMMs. A more serious concern is that replay might require logging a large amount of information and/or impose a high run-time overhead for the logging itself. However, results from [6] suggest that this overhead may be quite reasonable; a virtual machine running SPECweb99 produced 1.4 GB/day of log data per day, and the run-time overhead was less than 5%. Another concern is the cost of auditing and replay; we discuss this further in Section 3.4.

3.3 Trusted timestamping

Logging and replay can be used to detect incorrect executions, but this is just one of the problems that a cloud customer may be interested in detecting. SLA violations are likely to be another major concern; to detect these, we must add timing information to the tamper-evident log. For example, we can periodically (e.g., once per second) include in the log a certificate from a trusted third-party timestamping service, such as [1]. For obvious reasons, the timestamping service should neither be controlled by the customer nor by the cloud provider.

Once we have timing information in the logs, we can use it to detect performance faults. One way to do this is to replay the log on a machine of roughly the speed promised by the cloud provider, and to check whether the time required to replay a log segment between two timestamps t_1 and t_2 is at most $t_2 - t_1$. This is admittedly a rather crude method that can detect only very large deviations from the advertised speed. However, if the provider agrees to a more detailed specification (e.g., in terms of latency or throughput), it should be possible to check more fine-grain properties.

3.4 Sampling

If the customer is to detect faults by repeating every single step made by a cloud machine, he would need a second cloud to check the performance of the first, which seems impractical for most applications. One way to mitigate this problem is to use a conservative abstraction [5] and to check more coarse-grain properties. Another is to use sampling: We can achieve a probabilistic guarantee by having the cloud perform frequent checkpoints,[2] and by allowing the customer to randomly audit a small number of segments between consecutive checkpoints. Since many serious problems (such as hardware faults) will affect many or most of the segments, the customer can still detect them with high probability even if the sampling rate is low.

SLAs often include stochastic guarantees, and at first glance, sampling does not appear to be sufficient to audit these. For example, if the provider promises that 95% of response times will be less than 100 ms, how can the customer be sure that there is no fault without inspecting

[2] Since the size of the checkpoints depends on the amount of mutable state in the service, sampling is only efficient if this state is not too large.

all response times? The answer is that he cannot, but he can achieve arbitrarily high confidence by sampling a subset and by performing a χ^2 test. If the result is positive, the customer can confirm the presence of a fault by downloading and checking a larger log segment.

3.5 Challenges

The above set of techniques could be used to make a cloud accountable for correctness and, to a certain extent, for performance. However, it is not yet clear how to achieve accountability for other properties, such as confidentiality, at least not without the use of heavyweight primitives such as dynamic taint analysis [11]. Another difficult problem is support for services with legacy users, who may access cloud machines but do not maintain a tamper-evident log. However, it appears that solving this problem for certain classes of applications is feasible. For example, proxies could be used to add accountability to a legacy web service.

Clearly, another important concern is performance. Although there is evidence that the overhead will be manageable [6], we have yet to demonstrate that the cost of maintaining, transferring, and replaying the logs of a realistic cloud-based service is acceptable.

4 Related Work

Cachin et al. [4] contains a survey of security issues in the context of cloud storage services, and of recent research addressing these issues; Armbrust et al. [3] is a more general survey of cloud computing. Both of these papers point out some of the same challenges that motivate our work. Previous work has shown how to apply accountability to individual applications [7, 10, 14]; however, to the best of our knowledge, this paper is the first to propose accountability for an entire platform.

Trusted computing [12] is an alternative approach to achieving some of the guarantees we propose. However, it typically requires trusting the correctness of large and complex codebases, such as hypervisors, device drivers, or entire kernels, which are still beyond the reach of state-of-the-art verification techniques. In contrast, some forms of accountability have been implemented without special hardware and with very little trusted code. Other forms (such as accountability for data confidentiality) may require some platform support, but we expect that small and simple primitives comparable to TrInc [9] will be sufficient.

5 Conclusion

In this paper, we have proposed that clouds be made accountable to their customers, and we have argued that both the customers and the cloud providers stand to benefit – the former because they can check whether their computations are being performed correctly, and the latter because they can more easily handle complaints and resolve disputes. We have outlined requirements for an accountable cloud, and we have sketched a set of building blocks that can form the basis of an implementation.

This paper is intended as a call for action; clearly, much work remains to be done before accountable clouds can become a commercial reality. We believe, however, that accountability is a great opportunity for the cloud industry: it can mitigate risks for both the customer and the provider, and it can enable an entirely new range of cloud-based applications.

References

[1] Carlisle Adams, Pat Cain, Denis Pinkas, and Robert Zuccherato. RFC 3161: Internet X.509 public key infrastructure timestamp protocol (TSP). `http://tools.ietf.org/rfc/rfc3161.txt`, August 2001.

[2] Amazon Web Services. TC3 Health case study. `http://aws.amazon.com/solutions/case-studies/tc3-health/`.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report EECS-2009-28, University of California at Berkeley, February 2009.

[4] Christian Cachin, Idit Keidar, and Alexander Shraer. Trusting the cloud. *ACM SIGACT News*, 40(2):81–86, June 2009.

[5] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[6] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. OSDI*, December 2002.

[7] Andreas Haeberlen, Petr Kuznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. SOSP*, October 2007.

[8] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[9] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proc. NSDI*, Apr 2009.

[10] Nikolaos Michalakis, Robert Soulé, and Robert Grimm. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *Proc. NSDI*, April 2007.

[11] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. NDSS*, February 2005.

[12] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. Towards trusted cloud computing. In *Proc. HotCloud*, June 2009.

[13] Aydan R. Yumerefendi and Jeffrey S. Chase. Trust but verify: Accountability for internet services. In *ACM SIGOPS European Workshop*, September 2004.

[14] Aydan R. Yumerefendi and Jeffrey S. Chase. Strong accountability for network storage. *ACM Transactions on Storage*, 3(3):11, 2007.