Trusting the Cloud

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

Idit Keidar
Technion
Haifa, Israel
idish@ee.technion.ac.il

Alexander Shraer
Technion
Haifa, Israel
shralex@tx.technion.ac.il

Abstract

More and more users store data in “clouds” that are accessed remotely over the Internet. We survey well-known cryptographic tools for providing integrity and consistency for data stored in clouds and discuss recent research in cryptography and distributed computing addressing these problems.

Storing data in clouds

Many providers now offer a wide variety of flexible online data storage services, ranging from passive ones, such as online archiving, to active ones, such as collaboration and social networking. They have become known as computing and storage “clouds.” Such clouds allow users to abandon local storage and use online alternatives, such as Amazon S3, Nirvanix CloudNAS, or Microsoft SkyDrive. Some cloud providers utilize the fact that online storage can be accessed from any location connected to the Internet, and offer additional functionality; for example, Apple MobileMe allows users to synchronize common applications that run on multiples devices. Clouds also offer computation resources, such as Amazon EC2, which can significantly reduce the cost of maintaining such resources locally. Finally, online collaboration tools, such as Google Apps or versioning repositories for source code, make it easy to collaborate with colleagues across organizations and countries, as practiced by the authors of this paper.

What can go wrong?

Although the advantages of using clouds are unarguable, there are many risks involved with releasing control over your data. One concern that many users are aware of is loss of privacy. Nevertheless, the popularity of social networks and online data sharing repositories suggests that many users are willing to forfeit privacy, at least to some extent. Setting privacy aside, in this article we survey what else “can go wrong” when your data is stored in a cloud.

Availability is a major concern with any online service, as such services are bound to have some downtime. This was recently the case with Google Mail[1], Hotmail[2], Amazon S3[3] and MobileMe[4]. Users must also understand their service contract with the storage provider. For example, what happens if your payment for the storage is late? Can the storage provider decide that one of your documents violates its policy and terminate your service, denying you access to the data? Even the worst scenarios sometimes come true — a cloud storage-provider named LinkUp (MediaMax) went out of business last year after losing 45% of stored client data due to an error of a system administrator[5]. This incident also revealed that it is sometimes very costly for storage providers to keep storing old client data, and they look for ways to offload this responsibility to a third party. Can a client make sure that his data is safe and available?

No less important is guaranteeing the integrity of remotely stored data. One risk is that data can be damaged while in transit to or from the storage provider. Additionally, cloud storage, like any remote service, is exposed to malicious attacks from both outside and inside the provider's organization. For example, the servers of the Red Hat Linux distribution were recently attacked and the intruder managed to introduce a vulnerability and even sign some packages of the Linux operating-system distribution[6]. In its Security Advisory about the incident, Red Hat stated:

> *... we remain highly confident that our systems and processes prevented the intrusion from compromising RHN or the content distributed via RHN and accordingly believe that customers who keep their systems updated using Red Hat Network are not at risk.*

Unauthorized access to user data can occur even when no hackers are involved, e.g., resulting from a software malfunction at the provider. Such data breach occurred in Google Docs[7] during March 2009 and led the Electronic Privacy Information Center to petition[8] with the Federal Trade Commission asking to *"open an investigation into Google's Cloud Computing Services, to determine the adequacy of the privacy and security safeguards... "*. Another example, where data integrity was compromised as a result of provider malfunctions, is a recent incident with Amazon S3, where users experienced silent data corruption[9]. Later Amazon stated in response to user complaints[10]:

> *We've isolated this issue to a single load balancer that was brought into service at 10:55pm PDT on Friday, 6/20. It was taken out of service at 11am PDT Sunday, 6/22. While it was in service it handled a small fraction of Amazon S3's total requests in the US. Intermittently, under load, it was corrupting single bytes in the byte stream ... Based on our investigation with both internal and external customers, the small amount of traffic received by this particular load balancer, and the intermittent nature of the above issue on this one load balancer, this appears to have impacted a very small portion of PUTs during this time frame.*

A further complication arises when multiple users collaborate using cloud storage (or simply when one user synchronizes multiple devices). Here, consistency under concurrent access must be guaranteed.

[1] http://googleblog.blogspot.com/2009/02/current-gmail-outage.html

[2] http://www.datacenterknowledge.com/archives/2009/03/12/downtime-for-hotmail

[3] http://status.aws.amazon.com/s3-20080720.html

[4] http://blogs.zdnet.com/projectfailures/?p=908

[5] http://blogs.zdnet.com/projectfailures/?p=999

[6] https://rhn.redhat.com/errata/RHSA-2008-0855.html

[7] http://blogs.wsj.com/digits/2009/03/08/1214/

[8] http://cloudstoragestrategy.com/2009/03/trusting-the-cloud-the-ftc-and-google.html

[9] http://blogs.sun.com/gbrunett/entry/amazon_s3_silent_data_corruption

[10] http://developer.amazonwebservices.com/connect/thread.jspa?threadID=22709

A possible solution that comes to mind is using a Byzantine fault-tolerant replication protocol within the cloud (e.g., [14]); indeed this solution can provide perfect consistency and at the same time prevent data corruption caused by some threshold of faulty components within the cloud. However, since it is reasonable to assume that most of the servers belonging to a particular cloud provider run the same system installation and are most likely to be physically located in the same place (or even run on the same machine), such protocols might be inappropriate. Moreover, cloud-storage providers might have other reasons to avoid Byzantine fault-tolerant consensus protocols, as explained by Birman et al. [3]. Finally, even if this solves the problem from the perspective of the storage provider, here we are more interested in the users' perspective. A user perceives the cloud as a single trust domain and puts trust in it, whatever the precautions taken by the provider internally might be; in this sense, the cloud is not different from a single remote server. Note that when multiple clouds from different providers are used, running Byzantine-fault-tolerant protocols across several clouds might be appropriate (see next section).

What can we do?

Users can locally maintain a small amount of trusted memory and use well-known cryptographic methods in order to significantly reduce the need for trust in the storage cloud. A user can verify the integrity of his remotely stored data by keeping a short hash in local memory and authenticating server responses by re-calculating the hash of the received data and comparing it to the locally stored value. When the volume of data is large, this method is usually implemented using a hash tree [25], where the leaves are hashes of data blocks, and internal nodes are hashes of their children in the tree. A user is then able to verify any data block by storing only the root hash of the tree corresponding to his data [4]. This method requires a logarithmic number of cryptographic operations in the number of blocks, as only one branch of the tree from the root to the hash of an actual data block needs to be checked. Hash trees have been employed in many storage-system prototypes (TDB [22] and SiRiUS [13] are just two examples) and are used commercially in the Solaris ZFS filesystem[11]. Research on efficient cryptographic methods for authenticating data stored on servers is an active area [26, 28].

Although these methods permit a user to verify the integrity of data returned by a server, they do not allow a user to ascertain that the server is able to answer a query correctly without actually *issuing* that particular query. In other words, they do not assure the user that all the data is "still there". As the amount of data stored by the cloud for a client can be enormous, it is impractical (and might also be very costly) to retrieve all the data, if one's purpose is just to make sure that it is stored correctly. In recent work, Juels and Kaliski [18] and Ateniese et al. [2] introduced protocols for assuring a client that his data is retrievable with high probability, under the name of *Proofs of Retrievability* (PORs) and *Proofs of Data Possession* (PDP), respectively. They incur only a small, nearly constant overhead in communication complexity and some computational overhead by the server. The basic idea in such protocols is that additional information is encoded in the data prior to storing it. To make sure that the server really stores the data, a user submits challenges for a small sample of data blocks, and verifies server responses using the additional information encoded in the data. Recently, some improved schemes have been proposed and prototype systems have been implemented [29, 6, 5].

The above tools allow a single user to verify the integrity and availability of his own data. But when multiple users access the same data, they cannot guarantee integrity between a writer and multiple readers. Digital signatures may be used by a client to verify integrity of data created by others. Using this method, each client needs to sign all his data, as well as to store an authenticated public key of the others or the

[11] `http://blogs.sun.com/bonwick/entry/zfs_end_to_end_data`

root certificate of a public-key infrastructure in trusted memory. This method, however, does not rule out all attacks by a faulty or malicious storage service. Even if all data is signed during write operations, the server might omit the latest update when responding to a reader, and even worse, it might "split its brain," hiding updates of different clients from each other. Some solutions use trusted components in the system [11, 31] which allow clients to audit the server, guaranteeing atomicity even if the server is faulty. Without additional trust assumptions, the atomicity of all operations in the sense of linearizability [16] cannot be guaranteed; in fact, even weaker consistency notions, like sequential consistency [19], are not possible either [9]. Though a user may become suspicious when he does not see any updates from a collaborator, the user can only be certain that the server is not holding back information by communicating with the collaborator directly; such user-to-user communication is indeed employed in some systems for this purpose.

If not atomicity, then what consistency can be guaranteed to clients? The first to address this problem were Mazières and Shasha [24], who defined a so-called *forking* consistency condition. This condition ensures that if certain clients' perception of the execution becomes different, for example if the server hides a recent value of a completed write from a reader, then these two clients will never again see each other's newer operations, or else the server will be exposed as faulty. This prevents a situation where one user sees part of the updates issued by another user, and the server can choose which ones. Moreover, fork-consistency prevents Alice from seeing new updates by Bob and by Carol, while Bob sees only Alice's updates, where Alice and Bob might think they are mutually consistent, though they actually see different states. Essentially, with fork consistency, each client has a linearizable view of a sub-sequence of the execution, and client views can only become disjoint once they diverge from a common prefix; a simple definition can be found in [7]. The first protocol of this kind, realizing fork-consistent storage, was implemented in the SUNDR system [20].

To save cost and to improve performance, several weaker consistency conditions have been proposed. The notion of fork-sequential-consistency, introduced by Oprea and Reiter [27], allows client views to violate real-time order of the execution. The fork-* consistency condition due to Li and Mazières [21] allows the views of clients to include one more operation without detecting an attack after their views have diverged. This condition was used to provide meaningful service in a Byzantine-fault-tolerant replicated system, even when more than a third of the replicas are faulty [21].

Although consistency in the face of failures is crucial, it is no less important that the service is unaffected in the common case by the precautions taken to defend against a faulty server. In recent work [8, 7], we show that for all previously existing forking consistency conditions, and thus in the protocols that implement them with a single remote server, concurrent operations by different clients may block each other even if the provider is correct. More formally, these consistency conditions do not allow for protocols that are wait-free [15] when the storage provider is correct. We have also introduced a new consistency notion, called weak fork-linearizability, that does not suffer from this limitation, and yet provides meaningful semantics to clients [7].

One disadvantage of forking consistency conditions is that they are not so intuitive to understand as atomicity, for example. Aiming to provide simpler guarantees, we have introduced the notion of a Fail-Aware Untrusted Service [7]. Its basic idea is that each user should know which of his operations are seen consistently by each of the other users, and in addition, find out whenever the server violates atomicity. When all goes well, each operation of a user eventually becomes "stable" with respect to every other correct user, in the sense that they have a common view of the execution up to this operation. Thus, in all cases, users get either positive notifications indicating operation stability, or negative notifications when the server violates atomicity. Our Fail-Aware Untrusted Services rely on the well-established notions of eventual consistency [30] and fail-awareness [12], and adapt them to this setting. The FAUST protocol [7] implements this notion for a storage service, using an underlying weak fork-linearizable storage protocol. Intuitively, FAUST indicates stability as soon as additional information is gathered, either through the storage protocol, or whenever the clients communicate directly. However, all complete operations, even those not yet known to be stable, preserve causality [17]. Moreover, when the storage server is correct, FAUST guarantees strong safety (linearizability) and liveness (wait-freedom).

Obviously, if the cloud provider violates its specification or simply does not respond, not much can be done other than detecting this and taking one's business elsewhere in the future. It is, however, possible to be more prudent, and use multiple cloud providers from the outset, and here one can benefit from the fruitful research on Byzantine-fault-tolerant protocols. One possibility is running Byzantine-fault-tolerant state-machine replication, where each cloud maintains a single replica [10, 14]. This approach, however, requires computing resources within the cloud, as provided, e.g., by Amazon EC2, and not only storage. When only a simple storage interface is available, one can work with Byzantine Quorum Systems [23], e.g., by using Byzantine Disk Paxos [1]. However, in order to guarantee the atomicity of user operations and to tolerate the failure of one cloud, such protocols must employ at least four different clouds.

Summary

Though clouds are becoming increasingly popular, we have seen that some things can "go wrong" when one trusts a cloud provider with his data. Providing defenses for these is an active area of research. We presented a brief survey of solutions being proposed in this context. Nevertheless, these solutions are, at this point in time, academic. There are still questions regarding how well these protections can work in practice, and moreover, how easy-to-use they can be. Finally, we have yet to see how popular storing data in clouds will become, and what protections users will choose to use, if any.

References

[1] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Byzantine disk Paxos: Optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

[2] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proc. ACM CCS*, pages 598–609, 2007.

[3] K. Birman, G. Chockler, and R. van Renesse. Towards a cloud computing research agenda. *SIGACT News*, 40(2), June 2009.

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

[5] K. D. Bowers, A. Juels, and A. Oprea. Hail: A high-availability and integrity layer for cloud storage. Cryptology ePrint Archive, Report 2008/489, 2008. `http://eprint.iacr.org/`.

[6] K. D. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: Theory and implementation. Cryptology ePrint Archive, Report 2008/175, 2008. `http://eprint.iacr.org/`.

[7] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. In *Proc. DSN 2009, to appear. Full paper available as Tech. Report CCIT 712, Department of Electrical Engineering, Technion*, Dec. 2008.

[8] C. Cachin, I. Keidar, and A. Shraer. Fork sequential consistency is blocking. *IPL*, 109(7), 2009.

[9] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proc. PODC*, pages 129–138, 2007.

[10] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proc. OSDI*, pages 173–186, 1999.

[11] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proc. SOSP*, pages 189–204, 2007.

[12] C. Fetzer and F. Cristian. Fail-awareness in timed asynchronous systems. In *Proc. PODC*, 1996.

[13] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *Proc. NDSS*, 2003.

[14] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead Byzantine fault-tolerant storage. In *Proc. SOSP*, 2007.

[15] M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 11(1), 1991.

[16] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), 1990.

[17] P. W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enchance concurrency in distributed shared memories. In *Proc. ICDCS*, 1990.

[18] A. Juels and B. S. K. Jr. Pors: proofs of retrievability for large files. In *Proc. ACM CCS*, pages 584–597, 2007.

[19] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[20] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. OSDI*, 2004.

[21] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proc. NSDI*, 2007.

[22] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proc. OSDI*, 2000.

[23] D. Malkhi and M. K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[24] D. Mazières and D. Shasha. Building secure file systems out of Byzantine storage. In *Proc. PODC*, 2002.

[25] R. C. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[26] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *Trans. Storage*, 2(2):107–138, 2006.

[27] A. Oprea and M. K. Reiter. On consistency of encrypted files. In *Proc. DISC*, 2006.

[28] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *Proc. ACM CCS*, pages 437–448, 2008.

[29] H. Shacham and B. Waters. Compact proofs of retrievability. In J. Pieprzyk, editor, *Proceedings of Asiacrypt 2008*, volume 5350 of *LNCS*, pages 90–107. Springer-Verlag, Dec. 2008.

[30] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. SOSP*, 1995.

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