Clouds-of-Clouds for Dependability and Security: Geo-replication Meets the Cloud

Miguel Correia

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
miguel.p.correia@ist.utl.pt

Abstract. The complexity of large cloud offerings makes it extremely hard to guarantee their dependability and security. This paper extracts lessons from some years of research on the notion of using several clouds – instead of a single one– with the objective of achieving high dependability and security. We show that using such *clouds-of-clouds* it is possible for services to continue to operate correctly despite dependability and security issues in a subset of the clouds. We show this approach with three cases: clouds-of-clouds for storage with the DepSky system; cloud-of-clouds for data processing with MapReduce; the execution of arbitrary services in clouds-of-clouds with the EBAWA algorithm.

1 Introduction

The complexity of large cloud offerings makes it extremely hard to guarantee their dependability and security. In relation to dependability, that difficulty is clear from the outages suffered by essentially all major offerings in the past few years. Examples are the disruption of an Amazon EC2 region for almost one week in May 2011, and the February 29 2012 disruption of Windows Azure. These two problems alone left the two services much below the often promised availability of five nines. In both cases the problem was a subtle bug that led to a cascade of failures. Security events are less visible than outages, but several studies and news suggest that the problem is equally serious. An alarming case was that of a Google engineer who spied upon user Gmail/Gtalk communication. Other examples are the case of CyberLynk in which an ex-employee deleted a TV series from which there was no other copy (March 2011).

We have been exploring the notion of *clouds-of-clouds* as a solution to this chaos of accidental failures and security events. A cloud-of-clouds is a set of cloud offerings that are used by a consumer to run some service or application. A cloud-of-clouds is a virtual cloud, formed of real, physical, clouds. The notion is strongly related to the term *federated clouds* that also designates virtual clouds formed of real clouds [7]. However, the word "federation" suggests something created by the owners of the clouds, whereas cloud-of-clouds has a weaker semantics, usually associated to something created by cloud consumers: they can create their own cloud-of-clouds for having high security and/or dependability on top of cloud offerings they do not trust enough. Needless to say, the two abstractions

D. an Mey et al. (Eds.): Euro-Par 2013 Workshops, LNCS 8374, pp. 95–104, 2014.

involve solving some of the same problems, such as dealing with heterogeneity of management interfaces and access control mechanisms.

Clouds-of-clouds are an approach for dependability and security because they explore the *redundancy* and *diversity* provided by different cloud offerings, from different companies. The diversity of different clouds can avoid common mode failures, i.e., failures that affect the whole cloud, which was what led to the two above-mentioned disruptions. The bugs that led to the two disruptions did not exist in other clouds (to the best of our knowledge), so a virtual cloud formed of one of those plus other clouds would not fail.

This idea can be better understood by considering some of the dependability and security attributes [4]. A cloud-of-clouds can provide *availability* because most of the clouds will continue to provide their service even if there is a disruption in one (or a few) of them. It can provide *integrity* because most clouds will continue to provide correct data and execution even if one of them does not. A cloud-of-clouds can provide *disaster-tolerance* because clouds can be scattered through the globe, so most clouds will continue to operate even if one is physically destroyed by a natural disaster (earthquake, storm, tsunami, etc.). Finally, a cloud-of-clouds prevents *vendor lock-in* because services must support cloud diversity, although this is not exactly a dependability/security attribute.

We considerer that clouds can fail arbitrarily (Byzantine faults), not only by stopping delivering their service (crash faults). There is considerable evidence of the existence of *accidental Byzantine faults* that can corrupt data and software execution. Two examples: a study in Google datacenters found more than 8% DIMMs affected by errors yearly [18]; a Microsoft study found frequent CPU and core chipset faults in consumer PCs [13]. In relation to *malicious Byzantine faults*, a malicious insider with access to servers' management virtual machines is known to be able to obtain consumer passwords, private RSA keys, and files [16], thus to modify arbitrarily data and software executed in the cloud. The existence of malicious insiders is clear from the mentioned Google and CyberLynk cases.

This paper argues that clouds-of-clouds are a valid mechanism for cloud dependability and security. The argument is supported by three of our recent works[1]. The first – DepSky – exemplifies the use of clouds-of-clouds for *storage* [5]. The second shows the use of a cloud-of-clouds for *data processing* using MapReduce [9]. The third illustrates how clouds-of-clouds can be used to execute *arbitrary services* (e.g., file systems, databases, coordination services) in clouds-of-clouds using state machine replication and the EBAWA algorithm [20].

The three involve *replication* in different clouds. Although clouds-of-clouds might be used differently, this is the generic mechanism we consider in the paper. We use the term *geo-replication* because the individual clouds will be typically scattered geographically, interconnected by the Internet [15].

The paper starts by presenting opportunities and challenges of geo-replication for clouds-of-clouds (Section 2). Then, DepSky, BFT MapReduce, and EBAWA are discussed respectively in Sections 3, 4, and 5. Section 6 summarizes the lessons learned and concludes the paper.

[1] Full details about those works can be found on the papers here cited.

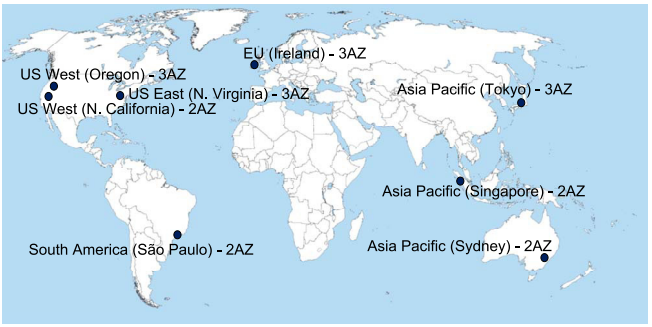

Fig. 1. Geographical redundancy and diversity of Amazon EC2's regions and availability zones

2 Opportunities and Challenges

This section discusses some of the opportunities and challenges of geo-replication for clouds-of-clouds taking Amazon's Elastic Cloud Computing (EC2) service as case of study. A cloud-of-clouds can, and from the dependability point of view should, be based on a set of different offerings from different providers to achieve best diversity. However, it is simpler to consider the case of a single company, which is also adequate as it has a world-wide operation. The discussion that follows is substantiated by a set of experimental data. We run one virtual server (an instance in Amazon's lingo) per Amazon region from August 2–15, 2013 (two weeks). We used Micro instances, i.e., the cheapest. We extracted round-trip times (RTT), throughput values, and information about Internet service providers (ISPs) and Autonomous Systems (ASs) crossed between every pair of nodes[2].

A first opportunity is the existence of *geographical redundancy and diversity*. This becomes clear in Figure 1 that shows Amazon EC2's 8 regions and the number of availability zones (AZs) per region (2 or 3). According to Amazon, "each region is completely independent" (which is coherent with the console that requires accessing one region at a time) and "each availability zone is isolated" [2]. The meaning of "independent" and "isolated" is not entirely clear but from the dependability point of view it suggests a considerable independence between AZs and a higher independence between regions. AZs in a region are connected through low-latency links, but regions are connected by the Internet.

A second opportunity is *network redundancy and diversity*. To understand this aspect, we run *lft* (a variation of *traceroute*) between every pair of instances we placed in EC2 regions and observed the ISPs and ASs crossed. The information about the ISPs for some of the pairs as of August 2nd, 2013, is shown in Figure 2 (we do not show all because they are many). All instances are in ASs from Amazon, which are not displayed, except in the case of the 3 US nodes that

[2] The raw data is available at `http://homepages.gsd.inesc-id.pt/~mpc/paper-data/amazon-ec2-aug2013.rar`

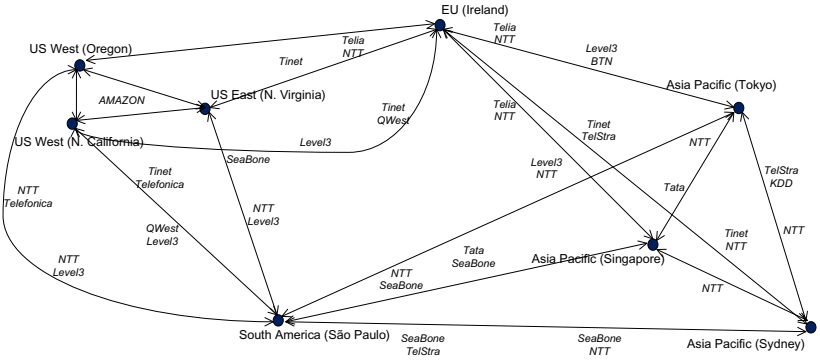

Fig. 2. ISP redundancy and diversity in some of the Amazon EC2 region interconnections in August 2nd, 2013

Table 1. Average and standard deviation of round-trip time (ms) and throughput (Mbps) between some pairs of instances placed in different Amazon EC2 regions

	nv-or	nv-nc	nv-ir	nv-si	nv-to	nv-sy	nv-sp	or-nv	or-nc	or-ir	or-si	or-to	or-sy	or-sp
RTT avg.	83.9	81.2	98.2	261.3	168.9	259.6	148.5	82.1	21.9	178.2	217.1	122.5	197.4	214.8
RTT std. dev.	14.2	13.3	11.3	11.3	12.4	9.2	26.2	5.0	6.6	8.2	12.3	8.5	23.2	9.2
Tput avg.	22.7	23.4	18.9	7.1	10.8	7.1	11.9	21.9	67.5	11.0	8.9	14.5	9.3	7.9
Tput std. dev.	3.0	2.8	3.7	1.2	1.8	0.9	2.8	4.1	20.5	2.5	1.3	2.6	1.7	1.6

are entirely connected by Amazon ASs. Generically it can be observed that several ISPs interconnect the regions, so there is diversity. Moreover, in most cases the ISPs crossed depend on the direction of the communication, indicated in the figure by two sets of ISPs in a link. For instance, when an IP datagram is sent from US West (Oregon) to South America (São Paulo) it crosses NTT and Level3; if sent the other way around it crosses Telefonica and NTT. This suggests the existence of a lot of redundancy. The ASs provide another level of diversity as most ISPs have several. These are not shown in the figure as it would become hard to read.

In relation to challenges, a first one is the *high and variant latencies* between sites. Table 1 shows the RTT between a few pairs of nodes. The regions are represented by the initials of the Country/State (e.g., *or* for Oregon). Even the two closest regions, US West (N. California) and US West (Oregon), show RTTs around 20 ms, which is 100 times more than, say, around 0.2 ms that can be observed in Ethernet LANs. The variation is also considerable.

A second challenge is the *low and variant throughput* between sites, as also observed in Table 1. It is important to note that Amazon provides better throughput to better (more expensive) instances, as shown in [19].

A third challenge is the *economic cost of sending data out of the cloud*. Amazon, as several other offerings, charges nothing for receiving data into its cloud from the Internet, only to send it out. The price for sending data out is zero for the first GB, then starts at 0.12 dollars per GB (up to 10 TB per month)

and goes down until 0.05 dollars per GB (for around 500 TB per month) [1]. The cost of sending data out is far from being the only cost of cloud computing: others are for instance the cost of storing data and the cost of running virtual machines (instances).

A fourth challenge is the CAP theorem that states the impossibility of having consistency, availability, and partition-tolerance at the same time [6,11]. The existence of high network diversity suggests that network partitions may not be an issue if best of breed clouds offerings like Amazon AWS are used. Nevertheless, partitions should probably be accounted for and a relaxed consistency may be offered when an partition actually occurs. This is a current topic of research that we, however, do not address more in this paper.

3 Storage – DepSky

DepSky is a software library for implementing storage clouds-of-clouds [5][3]. With DepSky, a consumer can implement a virtual storage cloud. Such a service provides file storage with operations such as create file, read, write, delete file, etc. A DepSky cloud-of-clouds is built on top of a set of legacy, unmodified, storage clouds. Many storage clouds can be used but currently the library has an abstraction layer that supports only four: Amazon S3, Google Storage, Rackspace Files, and Windows Azure Blob Storage. DepSky provides all the properties mentioned in the introduction – availability, integrity, disaster-tolerance, no vendor lock-in – plus confidentiality (file content cannot be disclosed at a cloud).

The core of DepSky are a pair of protocols for reading and writing files in the clouds. These protocols have to tolerate unresponsive clouds, so they have to reason in terms of quorums of clouds, i.e., of subsets of the group of clouds that implements the cloud-of-clouds. Moreover, these protocols have to tolerate Byzantine faults for the reasons mentioned in the introduction. Therefore, DepSky is based on Byzantine quorum (replication) protocols [12]. In the following discussion we consider n clouds at most f of which can fail. DepSky requires $n > 3f$ (e.g., $n = 4$ with $f = 1$).

The key idea of the protocols is simple. To *write a file* in the cloud-of-clouds, the client (i.e., the DepSky library at the user node) first writes the file in (at least) $n - f$ clouds, then writes a file with metadata in the same clouds. The metadata file contains information such as a cryptographic signature and a version. To *read a file*, the client first reads the metadata file from $n - f$ clouds, picks one of the metadata files with the highest version and gets the file from that cloud. If the signature does not correspond to the file retrieved, the file may have been corrupted, so the user retrieves the file from some of the other clouds.

This basic scheme has two limitations: files are stored in clear in all clouds, so confidentiality is worse than if stored in a single cloud; the storage used is n times more than if a single cloud was used, so the cost is on average also n times more. A solution to both problems is for each file: (1) to disperse it in n parts using erasure codes; (2) to generate a random secret key; (3) to encrypt

[3] Online at `https://code.google.com/p/depsky/`

each part of the file with a symmetric encryption algorithm (e.g., AES) and the key; (4) to split the key in n shares using a secret sharing algorithm; and (5) to store one pair {file part, key share} in each cloud. This way individual clouds (more precisely, a collusion of up to f clouds) cannot disclose the key or rebuild the file, so confidentiality is enforced. Moreover, storage space is approximately two times the size of the file.

We did an extensive experimental evaluation of DepSky using four commercial clouds and clients placed around the world in PlanetLab nodes. The more interesting results are the latency, i.e., the time to complete a read or write operation. We compared DepSky's latency with the latencies of the individual commercial clouds. On the negative side, we observed that DepSky's write latency is close to the one of the cloud with worst latency, which was to be expected as DepSky writes in all clouds (or at least $n - f$). DepSky's read latency is close to the latency provided by the cloud with lowest latency, which is good because clouds are often used to distribute content (i.e., more for reading than for writing files).

Our experience with DepSky provides several insights about the design of clouds-of-clouds: (1) The protocols should be based on Byzantine quorum systems to allow reasoning about subsets of clouds and tolerating Byzantine faults. (2) Signed files can be verified so they can be read from a single cloud, possibly the fastest, which typically provides the lowest latency (but not always). (3) Erasure codes are a valuable mechanism to reduce the size of data stored so also the cost involved. (4) Secret sharing can be used to store cryptographic keys in clouds (and avoid a key distribution scheme).

4 Data Processing – BFT MapReduce

Storage is probably the simplest service provided by clouds as it can be abstracted in terms of a simple set of operations: read, write, create, etc. Processing data is arguably more complicated. MapReduce is a programming model and an execution environment for processing large data sets made public by Google in 2004 [10]. *BFT MapReduce* is a modification of Hadoop MapReduce [21] for clouds-of-clouds that aims to provide the attributes mentioned in the introduction: availability, integrity, disaster-tolerance, no vendor lock-in [9]. However, on the contrary to DepSky and EBAWA, it tolerates only accidental Byzantine faults, not malicious Byzantine faults.

Hadoop MapReduce is built to tolerate the most common faults. The job tracker detects and recovers crashed map/reduce tasks and files are stored with checksums to allow detecting their corruption. However, Byzantine faults can corrupt task executions, so these tasks can return wrong outputs. Moreover, if a job is executing in a cloud and there is an outage, the job has to be restarted somewhere else (if the input splits are available somewhere else).

BFT MapReduce uses task replication to solve the first problem and replication in different clouds to solve the second. Consider again n clouds, at most f of which can stop or corrupt tasks (the original scheme is more generic and allows tasks to be corrupted in any cloud but this aspect complicates the presentation

so we omit it here). The scheme assumes that the splits are replicated in the n clouds and that $n > 2f$. Every split is processed by a map task in all clouds and all reduces are run in all clouds. The outputs of the replicas are voted and the result of the majority is accepted as correct (which is true due to the assumption of $n > 2f$). Even if a cloud fails the rest continue to execute the job.

This basic scheme has several performance and cost problems so we use three mechanisms to improve it. The first mechanism is to have one job tracker per cloud that controls the execution of tasks in its cloud. This solves the problem of controlling the execution of tasks from another cloud being a bad idea: the associated latency is high; the timeouts to detect if tasks failed have to be considerably higher (to limit wrong detections); the job tracker is a single point of failure.

The second mechanism is deferred execution: only $f + 1$ replicas of each task are executed as long as they all return the same output; when there is disagreement about the output, one or more additional replicas are executed (in other clouds). This reduces the resources needed to execute a job that have to be twice the original resources when there are no faults (if $f = 1$ then $f + 1 = 2$ replicas) or more (if $f > 1$).

The third mechanism is the one with the highest impact on the performance: digest communication. All reduce tasks have to fetch the outputs of all map tasks. When a reduce replica fetches the outputs of $f + 1$ replicas of the same map task it compares them and either accepts that output (if they are equal) or informs the job tracker that more replicas are needed. However, all map and reduce tasks are replicated so the number of outputs to fetch is typically large and fetching outputs through the Internet is slow. Moreover, these outputs can be large, for instance, of the same size of the inputs, therefore not only the delay but also the cost of taking this data out of a cloud will be high. The solution to this problem is to avoid transferring outputs between clouds: reduces only fetch cryptographic hashes (digests) of outputs from other clouds. These hashes are small (e.g., 32 bytes with SHA-256) and the comparison of hashes is equivalent to the comparison of the full outputs as they are collision-resistant [14].

Again our experience with BFT MapReduce provides several insights about the design of clouds-of-clouds: (1) Tasks can be replicated in different clouds to mask faulty executions and cloud failures. (2) Execution of some task replicas can be deferred to spare the resources used in some of the clouds. (3) Control components (the job tracker in this case) should be distributed to avoid control operations and failure detection between clouds and the associated high delays. (4) Whenever possible cryptographic hashes should be sent through the Internet instead of large messages, reducing communication latency and the cost of sending data out of the cloud.

5 Service Replication – EBAWA

Data storage (Section 3) and data processing using MapReduce (Section 4) are important but somewhat specific services for a cloud-of-clouds. *EBAWA* is a

Byzantine fault-tolerant state machine replication (SMR) algorithm for clouds-of-clouds [20][4]. SMR is a classical technique to make arbitrary services dependable (or fault-tolerant) through replication [17]. Example services that can be made dependable using SMR are databases, authentication services, coordination services, and file systems. Storage services and frameworks like MapReduce can also be made dependable this way but less efficiently than using the schemes presented in the previous two sections (e.g., more communication steps, more replicas). Similarly to BFT MapReduce, EBAWA provides availability, integrity, disaster-tolerance, and no vendor lock-in, but it tolerates both accidental and malicious Byzantine faults (as also does DepSky).

The idea of SMR is to replicate a service (a state machine) in n servers in such a way that if f become faulty the service continues operational (availability) and correct (integrity). For that to be possible the service has to be (or made) deterministic. Moreover, all replicas have to start in the same state and have to execute the same operations in the same order. The second aspect requires the use of a total order multicast protocol, which is the core of any SMR algorithm.

The first correct and efficient Byzantine fault-tolerant (BFT) SMR algorithm is known as PBFT [8]. This algorithm, however, was designed with LANs in mind. It has several communication steps (expensive when latency is high) and involves sending many messages. EBAWA is a BFT SMR algorithm like PBFT, but with a set of mechanisms for making it efficient in WANs, which make it adequate for clouds-of-clouds. In EBAWA replicas include a trusted module, the Unique Sequential Identifier Generator service (USIG). This is a local module that has to be implemented in such a way that it can be trusted, e.g., in hardware. The USIG service allows improving several of the characteristics of PBFT because it prevents certain kinds of faulty replica misbehavior: it prevents faulty replicas to send two different messages with the same identifier (an inconsistent fault). This allows to reduce the number of replicas from PBFT's $n > 3f$ to $n > 2f$. It also allows to reduce the number of communication steps by 1.

EBAWA differs from PBFT and similar algorithms by rotating its primary: the primary only orders one batch of requests, then the next replica becomes the primary. This prevents certain performance attacks [3], which are critical in WANs due to the need of using large timeouts (RTTs are much higher than in LANs). It reduces latency as clients can access the replica closest to them. Finally, it provides load balancing because the task of ordering messages is done by all replicas. EBAWA uses a third mechanism for efficiency in WANs: asynchronous views. The idea is that a replica starts an agreement as soon as it receives a client request by sending a prepare message. Replicas without pending client requests skip their turn by sending a special message.

We performed an extensive experimental evaluation of EBAWA in a real WAN (using PlanetLab), in an emulated WAN (Emulab), and in a LAN, showing interesting improvements of performance in comparison to PBFT.

[4] In fact EBAWA was designed before the trend on cloud computing so it was focused on the challenge of Byzantine fault-tolerant replication in wide-area networks. This, however, is the main challenge we are interested here in relation to clouds-of-clouds.

EBAWA provides several insights useful for the design of clouds-of-clouds: (1) The USIG service can be used to reduce the number of communication steps and improve the latency of the replicated service (by modifying the system model to include a trusted component). (2) The USIG service can be used to reduce the number of replicas, reducing the number of messages sent and the data sent out of the cloud (reducing the latency and the cost of sending data out). (3) Rotating the primary allows preventing performance attacks, client can access closest replica (reducing the latency), and load balancing. (4) Asynchronous views reduce waiting for the contacted replica to become the primary, so also the latency. (5) Waiting for $n - f$ replicas allows disregarding those with higher RTT, improving the latency.

6 Conclusion

The main argument of the paper is that clouds-of-clouds are a solution for consumers to create dependable and secure clouds on top of cloud offerings they do not trust enough. We have shown this with three cases: storage, data processing, and arbitrary services. A different question is if this solution is usable, due to its time and economic costs. Latency is an issue with clouds-of-clouds but it is also so with normal clouds. We have shown that it is possible to have cloud-of-cloud services with a latency of a few RTTs which is similar to what exists in a normal cloud (think of the delay of a simple HTTP access: 1.5 RTTs for the TCP 3-way handshake plus 0.5 RTTs for the reply, if we consider that the request goes with the last message of the handshake). In terms of costs, they are higher due to the use of more resources, but dependability and security are never free. Clouds-of-clouds seem to be a viable mechanism for dependability and security. If they will be adopted, the future will tell.

Acknowledgments. The three works discussed in this paper was done together with several colleagues and students: Fernando André, Alysson Bessani, Pedro Costa, Lau Cheuk Lung, Marcelo Pasin, Bruno Quaresma, Fernando Ramos, Paulo Sousa, Paulo Veríssimo, Giuliana Santos Veronese. This work was supported by national funds through FCT - Fundação para a Ciência e a Tecnologia, under project PEst-OE/EEI/LA0021/2013 (INESC-ID) and project PTDC/EIA-EIA/115211/2009 (RC-Clouds).

References

1. Amazon EC2 pricing, `http://aws.amazon.com/ec2/pricing/` (accessed August 15, 2013)
2. Amazon EC2 user guide – regions and availability zones, `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html` (accessed August 15, 2013)
3. Amir, Y., Coan, B., Kirsch, J., Lane, J.: Byzantine replication under attack. In: Proceedings of the IEEE/IFIP 38th International Conference on Dependable Systems and Networks, pp. 197–206 (June 2008)

4. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004)
5. Bessani, A., Correia, M., Quaresma, B., André, F., Sousa, P.: DepSky: Dependable and secure storage in a cloud-of-clouds. In: Proceedings of the 6th ACM SIGOPS/EuroSys European Systems Conference, pp. 31–46 (April 2011)
6. Brewer, E.: Towards robust distributed systems. In: Proceedings of the 19th ACM Annual Symposium on Principles of Distributed Computing, pp. 7–10 (2000)
7. Cascella, R.G., Blasi, L., Jegou, Y., Coppola, M., Morin, C.: Contrail: Distributed application deployment under SLA in federated heterogeneous clouds. In: Galis, A., Gavras, A. (eds.) FIA 2013. LNCS, vol. 7858, pp. 91–103. Springer, Heidelberg (2013)
8. Castro, M., Liskov, B.: Practical Byzantine fault tolerance. In: Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation, pp. 173–186 (February 1999)
9. Correia, M., Costa, P., Pasin, M., Bessani, A., Ramos, F., Verissimo, P.: On the feasibility of Byzantine fault-tolerant MapReduce in clouds-of-clouds. In: Proceedings of the 1st International Workshop on Dependability Issues in Cloud Computing (October 2012)
10. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation, pp. 137–150 (December 2004)
11. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, 51–59 (June 2002)
12. Malkhi, D., Reiter, M.: Byzantine quorum systems. Distributed Computing 11(4), 203–213 (1998)
13. Nightingale, E.B., Douceur, J.R., Orgovan, V.: Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs. In: Proceedings of the 6th ACM SIGOPS/EuroSys European Systems Conference, pp. 343–356 (2011)
14. NIST: FIPS 180-2, Secure Hash Standard (August 2002)
15. Rabinovich, M., Rabinovich, I., Rajaraman, R., Aggarwal, A.: A dynamic object replication and migration protocol for an internet hosting service. In: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, pp. 101–113 (1999)
16. Rocha, F., Correia, M.: Lucy in the sky without diamonds: Stealing confidential data in the cloud. In: Proceedings of the 1st International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments (2011)
17. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys 22(4), 299–319 (1990)
18. Schroeder, B., Pinheiro, E., Weber, W.D.: DRAM errors in the wild: a large-scale field study. In: Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems, pp. 193–204 (2009)
19. Topchiy, S.: Testing Amazon EC2 network speed (March 2013), `http://epamcloud.blogspot.pt/2013/03/testing-amazon-ec2-network-speed.html` (accessed August 15, 2013)
20. Veronese, G.S., Correia, M., Bessani, A.N., Lung, L.C.: EBAWA: Efficient Byzantine agreement for wide-area networks. In: Proceedings of the IEEE 12th International Symposium on High-Assurance Systems Engineering, pp. 10–19 (November 2010)
21. White, T.: Hadoop: The Definitive Guide. O'Reilly (2009)