Development of the Lock Protocol for DEPSKY Storage System

Poliana S. Nascimento, *Student in Technology for Systems Analysis and Development, IFBA*,
Allan E. S. Freitas, *Doctor in Computer Science, IFBA*

***Abstract*—Data management in environments based on several clouds (cloud-of-clouds) should be dependable and secure. DEPSKY may assure that characteristics through mechanisms as cryptography and data replication, however DEPSKY does not support concurrent writing, a desirable functionality for many applications. This paper presents the development and a performance analysis of a lock algorithm for DEPSKY storage system. The paper also presents validation and performance tests of the algorithm. Such protocol allows concurrent writing, through a low contention lock mechanism that uses lock files to define who is allowed to write in a data unit.**

***Index Terms*—Cloud Computing, Replication, Security.**

I. Introduction

The growing use of computing environments on the Internet and that need for sharing information and resources makes very important the use of cloud computing environments [1]. In those environments, given the need for resource sharing, there is a growing concern about the security and privacy of the stored data [2].

Several works in the literature studies the problem of cloud storage. For instance, RACS (*Redundant Array of Cloud Storage*) is a transparent proxy for cloud storage that distributes the load for many service providers [3]. The proposed algorithm is fault tolerant and reduces the vendor lock-in [1], making it easier to migrate between platforms. Nevertheless, it does not deal with security issues.

There is also HAIL (*High-Availability and Integrity Layer*), which is a criptographic distributed system that manages the integrity of the files and their availability through a collection of servers or cloud storage services [4]. Nevertheless, it has some limitations such as the need for servers to execute code, the lack of mechanisms to protect the confidentiality of the stored data and the fact that it works only with static data [5].

DEPSKY is a reliable and safe storage system that manages data stores in clouds based on different commercial cloud services in a cloud-of-clouds environment [6]. This way, DEPSKY offers facilities to go around important limitations in cloud data storage, such as loss of availability, data loss and alteration, privacy loss and vendor lock-in. In opposition of the previously described systems, DEPSKY deals with security issues, and it has mechanisms to protect stored data confidentiality and does not require that servers need execute additional code.

According to the CAP theorem (Consistency, Availability, Partition tolerance) [7], a distributed database may fulfill simultaneously only two of these three properties: Consistency, the guarantee that all data is update in all replicas; Availability, that is, in case a node fails, there is at least one able to answer the requisitions; and Partitioning tolerance, the ability to continue working even when all copies cannot communicate among themselves.

This way, DEPSKY can provide a relaxed consistency, given the possibility of network partitioning [8], that may not be a problem in scenarios such as those where applications do not require to handle sensible data and that require low response time, such as, for instance, social network applications. Using adequate commercial cloud platforms, such as *Amazon S3*, may decrease the possibility of this partitioning. This discussion is not the focus of this paper.

Nevertheless, the DEPSKY implementation did not support multiple writers, which is a desirable feature for full use of cloud-of-clouds environments. This way, this paper presents the development of a lock algorithm, as proposed by [6]. Since most fault tolerance systems that will use DEPSKY will only have a node, there is no need for a high level concurrency protocol. Hence, it is proposed a low contention level lock mechanism using blocking files to define who is the writer in the data unit, allowing for concurrency.

The developed algorithm was evaluated measuring a commercial cloud service, the *Amazon S3* [9]. This way, it was possible to test and evaluate the performance of the system when multiple clients are writing simultaneously.

This paper is organized as follows. Section 2 will make a brief introduction on DEPSKY, and these protocols:

Corresponding author: Allan E. S. Freitas, *allan@ifba.edu.br*. This implementation work for the proposed code and its performance evaluation was developed by the student Poliana dos Santos Nascimento in a research study during the interchange program at the LaSIGE group at the Science College at the University of Lisbon, under the guidance of professor Alysson Bessan and tutorship of PhD student Tiago Oliveira.

[1]The vendor lock-in is related to the technological dependency on the used platform, which makes it difficult to move clients from one vendor to another, making some vendors dominants.

reading, writing, lock and unlock; and some aspects of the development of lock and unlock algorithms. Section 3 shows an analysis of the performance and finally, Section 4 presents some final considerations.

II. DEPSKY and the development of lock and unlock algorithms

DEPSKY [6] is a virtual storage system that replicates the data in several clouds, improving availability, integrity and confidentiality of the stored information. It uses Byzantine quorums, secret sharing and erasure codes in order to ensure fault tolerance[2] and confidentiality. This way, users can manage the data by invoking operations in the different individual clouds. Figure 1 presents an example of DEPSKY architecture configuration, where two clients are in communication with four servers, each one situated in a distinct cloud.

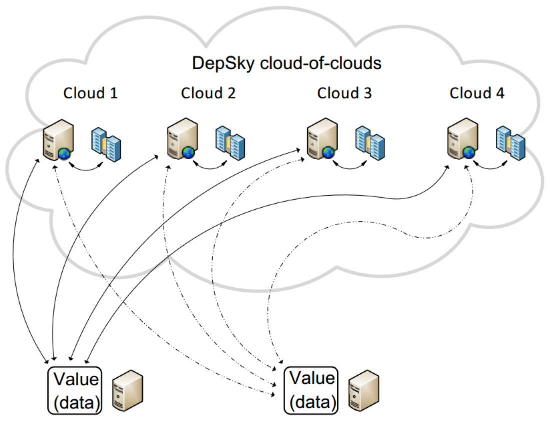


Fig. 1. DEPSKY architecture, with fours clouds and two clients [6].

This systems protects most sensitive information using a secret sharing mechanism. For that, it used two protocols that differ in the way the information is sent to the clouds. In the ADS protocol (Available DEPSKY) a copy of the information is sent to all clouds. In the CADS protocol (Confidential and Available DEPSKY) the information is broken into parts according to the number of clouds and each part is sent to a cloud. The information is codified such that only a certain number of parts is need to rebuild the original information, instead of all parts [6][5].

A. *Reading and writing protocols*

The writing protocol present in DEPSKY is based on the fundamental idea of keeping he value in a quorum of clouds (data storage units), and then recording the corresponding meta-data assuring that only one writer (also known as client, the process that wants to block the cloud in order to trigger the writing mechanism) is able to read the meta-data for the stored value. When a client writes the first data unit (basic storage unit for the algorithm, which contains a unique name, a version number and a verification date and the data stored in the object), the client gets in touch with all the clouds in order to get the meta-data with the higher version number, updating the variable that stores the version number [6].

The DEPSKY reading protocol gets the meta-data files in a cloud quorum and chooses the one with the higher version number. Afterwards, it searches the version of the data unit that corresponds to this version number and the criptographic hash found in the metadata. Afterwards, the protocol gets the file that contains the correspondig value to the matedata synthesis in the cloud quorum. If those conditions are satisfied, the process leaves the loop and returns the value [6].

B. *Lock and unlock protocols*

The developed lock protocol is a blocking mechanism for low contention, based in a file that specifies who is the writer and how long it is allowed writing in the data unit. This is illustrated in Figure 2, which presents the lock and unlock algorithms. The algorithms work as follows: when a process wants to become a writer, it lists the file in the cloud and looks for a file named *lock-c'-T'*, in which *c'* is the identifier of the associated process and *T'* is the duration of time for which this process received permission for writing (lines 5-10). If this file is found in the data unit, it means that some process is blocking it, hence the process c hibernates for a random time (line 21). If it is not found, c may write the blocking file, including a cryptographic signature of the file name in all clouds (lines 11 and 12). The withdrawal strategy is necessary to ensure that two or more processes do not receive authorization to write at the same time [6].

Finally, the process c lists again all the files in the cloud, searching for blocking files (lines 13-17). If he finds a file that is not his with a expired writing time, c removes its lock file and hibernates for a given period of time [3]. If he does not find any lock file, c becomes the only writer for the data unit. The unlock is performed through the removal of the lock file through the unlock protocol(lines 24-27) [6].

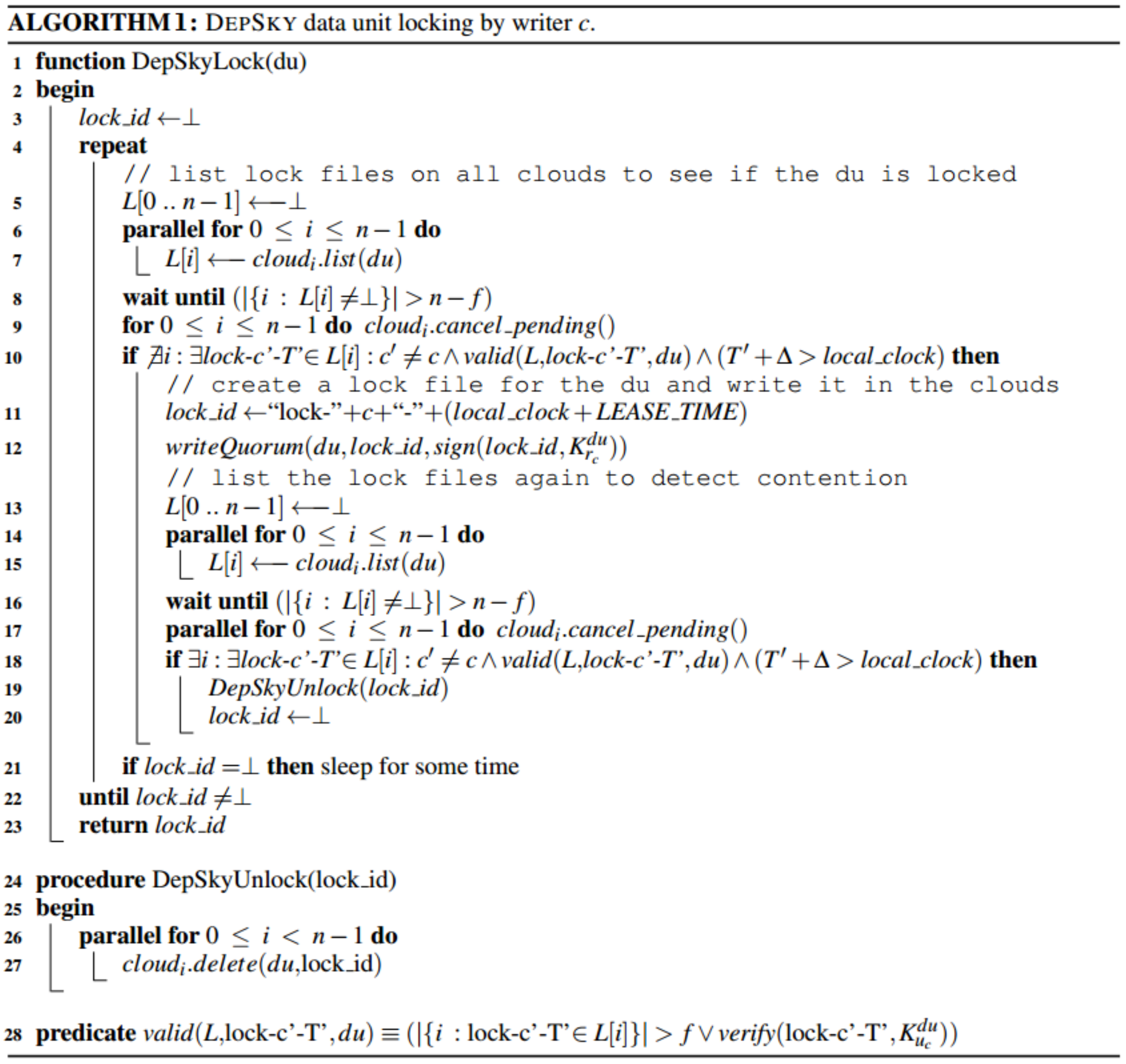
It is important to observe that the unlock procedure is not completely fault tolerant. In order to liberate the writing of a lock file, it must be deleted from all the clouds, but a single cloud may fail, keeping the file that should have been removed, if the unlock function fails during the exclusion. This will deny the block by other writers, but since every blocking file has an expiration, this problem will only affect the system for a specific period of time, because the file will expire [6].

C. *Aspects of the algorithm development*

Lock and unlock algorithms were implemented in its corresponding functions in the JAVA programming language, as well as all DEPSKY, because JAVA provides

[2]DEPSKY tolerates Byzantine faults, and hence it can tolerate the failure of up to f of the n clouds, where $n > 3f$.

[3]This period of time is defined randomly at run-time according to a time limit predefined in the application.



ALGORITHM 1: DepSky data unit locking by writer c.

```
function DepSkyLock(du)
begin
    $lock\_id \leftarrow \perp$
    repeat
        // list lock files on all clouds to see if the du is locked
        $L[0 .. n-1] \longleftarrow \perp$
        parallel for $0 \leq i \leq n-1$ do
            $L[i] \longleftarrow cloud_i.list(du)$
        wait until $(|\{i : L[i] \neq \perp\}| > n-f)$
        for $0 \leq i \leq n-1$ do $cloud_i.cancel\_pending()$
        if $\nexists i : \exists lock\text{-}c'\text{-}T' \in L[i] : c' \neq c \wedge valid(L, lock\text{-}c'\text{-}T', du) \wedge (T' + \Delta > local\_clock)$ then
            // create a lock file for the du and write it in the clouds
            $lock\_id \leftarrow$ "lock-"$+c+$"-"$+(local\_clock + LEASE\_TIME)$
            $writeQuorum(du, lock\_id, sign(lock\_id, K^{du}_{r_c}))$
            // list the lock files again to detect contention
            $L[0 .. n-1] \longleftarrow \perp$
            parallel for $0 \leq i \leq n-1$ do
                $L[i] \longleftarrow cloud_i.list(du)$
            wait until $(|\{i : L[i] \neq \perp\}| > n-f)$
            parallel for $0 \leq i \leq n-1$ do $cloud_i.cancel\_pending()$
            if $\exists i : \exists lock\text{-}c'\text{-}T' \in L[i] : c' \neq c \wedge valid(L, lock\text{-}c'\text{-}T', du) \wedge (T' + \Delta > local\_clock)$ then
                $DepSkyUnlock(lock\_id)$
                $lock\_id \leftarrow \perp$
        if $lock\_id = \perp$ then sleep for some time
    until $lock\_id \neq \perp$
    return $lock\_id$

procedure DepSkyUnlock(lock_id)
begin
    parallel for $0 \leq i < n-1$ do
        $cloud_i.delete(du, lock\_id)$

predicate $valid(L, lock\text{-}c'\text{-}T', du) \equiv (|\{i : lock\text{-}c'\text{-}T' \in L[i]\}| > f \vee verify(lock\text{-}c'\text{-}T', K^{du}_{u_c}))$
```

Fig. 2. Lock and unlock algorithms [6].

many libraries and services that make it easier to read and write at the used clouds [5] [10]. As auxiliary functions for this algorithm, we developped the functions *listQuorum*, *writeQuorum* and *deleteData*[4].

The function *listQuorum* lists all the lock files in the clouds. It is used by the *lock* function to bring up the list of block files. The *writeQuorum* functions writes in the clouds the lock file, using a SHA-1 cryptographic signature of the name of the file that contains the time-stamp from the moment the process was able to perform the block. The *deleteData* function is used by the *unlock* function and removes the lock file from the clouds. All messages are transmitted using symmetric cryptography using the AES algorithm.

The *unlock* functions removes the blocking file, releasing the clouds for other processes to acquire the writing access. For the *unlock* function, it was necessary to perform some changes in the original algorithm from the Figure 2. Instead of passing only the *lock_id*, it was necessary to also pass the (*DataUnit*), because the function *deleteData* demands this parameter to be passed to perform the exclusion.

In the function *lock* [5] the *DataUnit* is passed as a parameter, as well as the number of times the process wants to try to block the clouds. This function verifies if a process is blocking the data unit. First, it calls the function *listQuorum* to verify if there is a lock file in the cloud. If it returns *null*, the function *writeQuorum* is called. Afterward, the function *listQuorum* is called again to verify if there is a lock file besides the one written by the client. If another process wrote in the cloud, the client removes its lock file and hibernates for a random time. If there is no other file, it becomes the single writer for this data unit.

[4]The code for the functions *listQuorum*, *writeQuorum* and *deleteData* are available at the Internet address https://code.google.com/p/depsky/source/browse/trunk/DepSky/src/depskys/core/LocalDepSkySClient.java

[5]The codes for the functions *lock* and *unlock* are available at the Internet address https://code.google.com/p/depsky/source/browse/trunk/DepSky/src/depskys/core/LocalDepSkySClient.java

If the list is not empty, it is verified if the file kept in the cloud is for the selected data unit, if it belongs to the process that is trying to write and if the cryptographic signature is valid. If all conditions are satisfied, the data unit is selected and the process becomes the single writer. If not, it verifies if the blocking time for the file in the cloud expired and if so, the process removes the lock file from the cloud and writes its own. Otherwise, the process hibernates for a random time.

Some observations must be made on the protocol and its implementation. Block should be renewed periodically to guarantee the existence of a single writer during the execution. If several process try to become writers at the same time, it is possible that none of them became the writer. Given the fallback strategy used, we assume as premise that this condition may not be so frequent [6] – and even in this case, the clients may resubmit afterward their writing requests.

The protocol does not guarantee termination. When a client requests the lock, a variable is passed as parameter to inform the number of times the process will try to get the lock and this guarantee that it does no try endlessly and fruitlessly, avoiding starvation. Nevertheless, the property of progress is not guaranteed, because there is a limited amount of locking attempts and so the writing may be unsuccessful. The lock protocol assumes as a premise that the clocks are periodically synchronized.

Figure 3 illustrates the working of the lock protocol for the case a client succeed into become the writer for the data unit at the first attempt. First, the client requests the clouds to return a list of lock files. Afterward, it writes the lock file and requests again the list of lock files to verify it another client tried to write at the cloud. After this procedure, it uses the write protocol and finalizes, calling the *deletedata* function to unlock the cloud.

III. Performance analysis

THIS section deals with aspects of the performance analysis.

A. Experimentation environment

The client was executed in a desktop computer with the *GNU/Linux Ubuntu 12.0* operating system, *Intel Core i5* processor and 4 GB RAM memory. For data storage, we created 4 buckets at *Amazon S3*[6] [9]. Each of these buckets were created in different locations: São Paulo, Ireland, Stansted and USA West.

The different locations of the buckets assured failure independence. The amount of at least 4 buckets is necessary to provide a quorum that is tolerant to at most one Byzantine failed bucket (i.e. $N > 3F$).

In order to perform, the tests we collected data referring to the time for lock, write, unlock operations and the total time. For these measurements, we considered the time each client invokes the write operation followed by the unlock operation, after it succeed in writing the lock file in the cloud.

B. Analyzed criteria

- *Writing time at the clouds*: We analyzed the time a process takes to perform the lock, write and unlock operations. We observed the time spent confronting the increase of the number of process that tries to perform the writing operations simultaneously at the clouds. First the client starts the lock protocol and in case the cloud is blocked it hibernates and if not, it requests the cloud to be blocked and afterward writes and unlock the cloud, liberating it for other clients to access it.
- *Process failures*: In this analysis, a process fails to block the cloud and does not trigger the unblocking mechanism, stopping other processes that try to execute the lock protocol. We also conducted experiments to verify whether the process can identify if a lock file actually belongs to it.
- *Cloud failure for invasion or corruption of the data*: In this analysis, the cryptographic signature of more than f clouds were modified in an incorrect way, where f is the maximum amount of clouds in error according to the used quorum. If the process identifies the problem, it should delete the lock file to release the cloud.

C. Results and discussion

1) Time to write in the cloud: In this analysis, we used as factor the number of processes (1, 2, 3 and 4). We defined as parameters the number of attempt the client tried to execute the locking mechanism (20 attempts), a number defined *ad-hoc* to minimize the possibility the clients will finish their execution with no success; and the hibernation time a process takes when waiting its turn, which is determined as a fixed value ($1000ms$) added to a random value.

The cumulative distribution graphics for the lock and unlock operations and for the total time are presented in Figures 4, 5 and 6, respectively, exposing the accumulated frequency (vertical axis) for a certain operation with the observed latency in seconds (horizontal axis). The graphs allow for a comparison of the latency according to the increase of the number of clients. The value of the frequency equals to 1 indicates that all operations happen in a time up to the indicated latency. Each line represents the variation of the latency according to the number of clients.

It was possible to observe that when the number of clients increase, the total time required to finish the operation also increases, which implicates in a loss of performance, given that the more clients that try to access the cloud, higher the time each process takes to finalize writing.

[6]*Amazon S3 – Simple Storage Service* – is a PaaS cloud that provides data storage services in the form of units called Buckets through a simple web interface that can be used to store and recover any amount of data at any moment and any given web location [9]

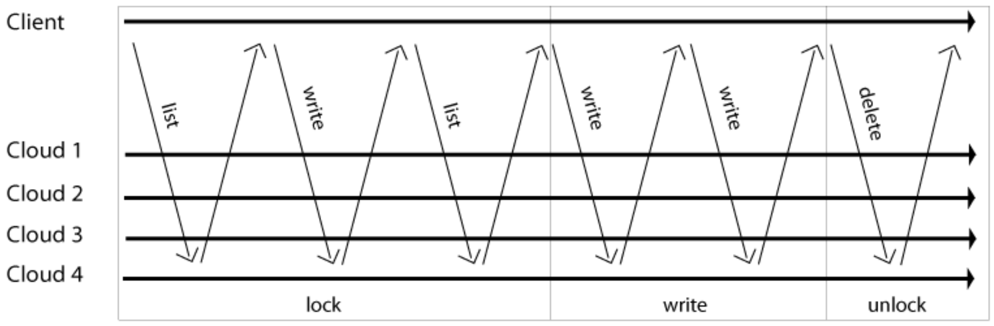


Fig. 3. Writing process using the lock protocol.

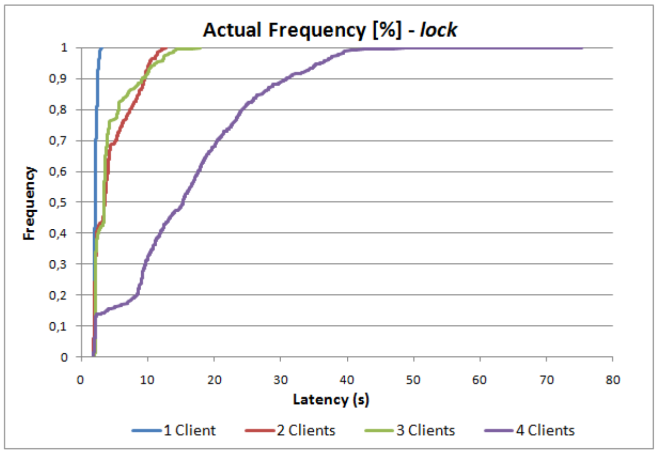


Fig. 4. Cumulative distribution graphic with one, two, three and four clients – lock operation.

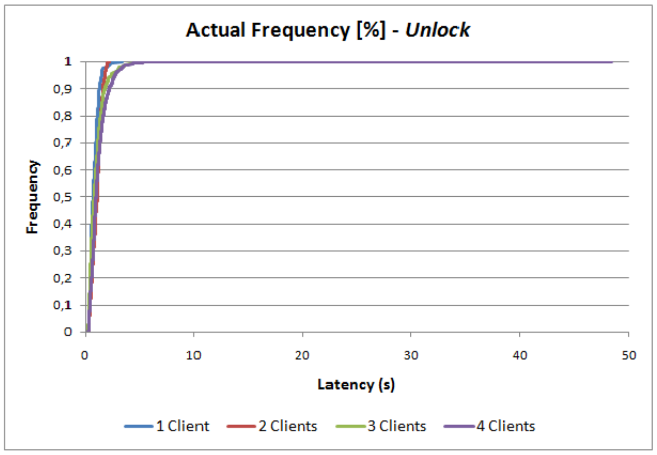


Fig. 5. Cumulative distribution graphic with one, two, three and four clients – unlock operation.

Figure 4 indicates the performance loss at the lock operation as a consequence of the increase of the number of clients that tried to block the cloud. We observed that the latency increased with the number of processes that were competing to write at the data unit. We analyzed that as the number of clients increased, even then operations that were not competing to write at the cloud, such as write and unlock, became more idle, since more than one process was trying to access the cloud, causing lack of efficiency, as can be seen in Figure 5 at the cumulative distribution graphics for the unlock operation.

When analyzing the cumulative distribution graphic for the unlock poeration presented in Figure 5, it was possible to realize that the time for the unlock operation varied within a long tolerable margin, growing gradatively with the number of clients. The performance of the lock operation fell more rapidly that the unlock one, because of the concurrency to perform the block of the clouds by the clients, which did not occurr inthe unlock protocol, given that the client ha alrady become the writer.

Figure 6 presetns the current frequency graphic for the total time for the three operations (lock, write e unlock). As previously discussed, it is easy to realize that the latency increases with the number of clients. But it is important to report that using the lock protocol, the time to perform the writing at the data unit increases, because we used three operations to perform the process in which the writers enter in a race condition. This caused an increase in functionality for the system, but on the other hand, there was a loss in efficiency.

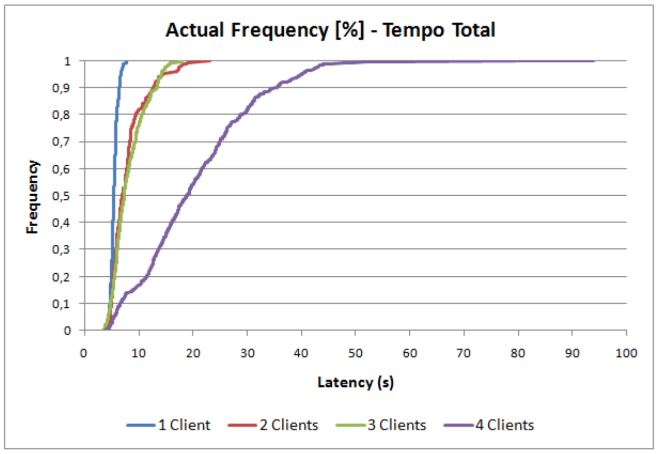


Fig. 6. Cumulative distribution graphic with one, two, three and four clients – total time for the operations lock, write and unlock.

Figure 6, presents the current frequency of the total time taken for all operation and as suc, allow for a better comparison of the results found at the experiment with one, two, three and four clients.

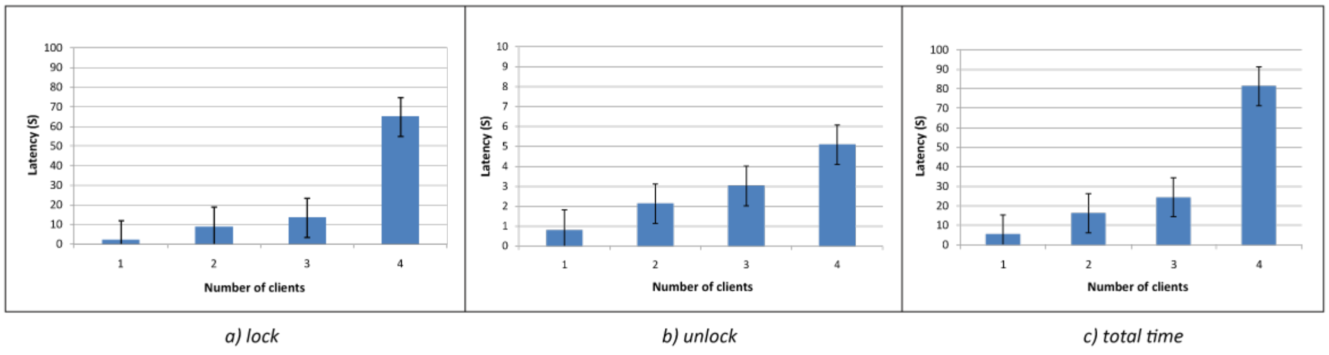


Fig. 7. Graphics comparing one, two, three and four clients - lock and unlock operations and total time.

The loss of performance caused by the number of process that are competing to write may be simplified by Figure 7, which presents a comparision of the latency as a consequence of the increase of the number of writer for the operations lock and unlock and for the total process time. The graphics also show the standard deviation. Finally, it was observed that the total time increased was caused primarily by the competition for the processes to write the lock file at the data unit.

2) Failure of the process: In this analysis, we studied how the lock protocol behaved when failures to write occurred, making it impossible to trigger the unlock mechanism. We tested two executing process. One of them failed when writing at the cloud while the other tried to block the cloud but was unsuccessful because there was a lock file at the cloud. In this situation, it was possible tho verify that given the lock protocol principle that a process can only block the cloud for a given period of time[7], even when the client does not unlock, when its time expires, the other process can remove the lock file, unlocking the cloud.

3) Failure in the cloud because of invasion and corruption of data: In this analysis, we simulated an arbitrary behavior in which two of the four clouds executed maliciously, what was verified by the process through the criptographic signature informed by the clouds. It must be noticed that in this execution the Byzantine quorum was not respected, since only one of the clouds may fail so that the requisites are fulfilled. Given that only two clouds answered with the correct file, we did not satisfy the quorum of at least $2f + 1$ clouds and as expected, the client excluded the lock file stored in the clouds.

IV. Conclusion

This paper present the development and the analysis of the performance of a lock algorith for the DEPSKY storage system. The lock protocol allows for more than one client to write at the data unit through a lock mechanism of low contention using a file that stores the information of who is the current writer at the data unit.

[7]This period of time was defined in a random way during runtime, according to a time limit predefined in the application.

Analyzing the results found, it was possible to observe that the developed algorithm satisfies the contions to perform the lock of the clouds, stopping more than one client to write at the same time at the data unit. Besides, it was possible to verify that the performance degrades given the simultaneous write: the higher the number of clients that try to get writing access to the clouds, the higher the service overload and the latency observed.

References

[1] Q. Zhang, L. Cheng, and R. Boutaba, “Cloud computing: state-of-the-art and research challenges”, *Journal of internet services and applications*, vol. 1, no. 1, pp. 7–18, 2010.

[2] E. Hanna, N. Mohamed, and J. Al-Jaroodi, “The cloud: requirements for a better service”, *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 787–792, 2012.

[3] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, “Racs: a case for cloud storage diversity”, in *Proceedings of the 1st ACM symposium on Cloud computing*, ACM, 2010, pp. 229–240.

[4] K. D. Bowers, A. Juels, and A. Oprea, “Hail: a high-availability and integrity layer for cloud storage”, in *Proceedings of the 16th ACM conference on Computer and communications security*, ACM, 2009, pp. 187–198.

[5] B. M. M. R. Quaresma, “Depsky: sistema de armazenamento em clouds tolerante a intrusões”, Master’s thesis, UNIVERSIDADE DE LISBOA. Faculdade de Ciências. Departamento de Informática, 2010.

[6] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, “Depsky: dependable and secure storage in a cloud-of-clouds”, *ACM Transactions on Storage (TOS)*, vol. 9, no. 4, p. 12, 2013.

[7] L. Frank, R. U. Pedersen, C. H. Frank, and N. J. Larsson, “The cap theorem versus databases with relaxed acid properties”, in *Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication*, ACM, 2014, p. 78.

[8] M. Correia, “Clouds-of-clouds for dependability and security: geo-replication meets the cloud”, in *Euro-Par 2013: Parallel Processing Workshops*, Springer, 2014, pp. 95–104.

[9] *Amazon web services. available at https://console.aws.amazon.com/console/home.*

[10] *Depsky: a cloud-of-clouds storage middleware. available at https://code.google.com/p/depsky/.*