EVALUATION OF THE PERFORMANCE OF TWO PUBLIC CLOUD COMPUTING PLATFORMS THROUGH THE USE OF A DISTRIBUTED ENCRYPTION SYSTEM

Erich Rice & Dennis Guster
Information Systems
St. Cloud State University
St. Cloud, MN 56301
eprice@stcloudstate.edu
dcguster@stcloudstate.edu

Richard Anderson
Alumni
Information Systems
St. Cloud State University
St. Cloud, MN 56301
richardalmanderson@gmail.com

ABSTRACT

Cloud computing has become nearly as ubiquitous as the Internet in recent years, with large Cloud computing providers such as Amazon, Microsoft, IBM, and Google dominating much of the marketplace for public Cloud services. Both individual users as well as enterprises of every size have moved their data and computing needs into the Cloud. Often these providers promise a lower price point than on-premises solutions. In this paper, the main focus will be on the relative speed a Cloud-based system will work over the Internet. In this paper two different Cloud computing platforms are compared for performance the Amazon Web Service (AWS) and DigitalOcean. Both Cloud providers are based in the United States but have multiple datacenters located around the globe. The method by which the two Clouds will be compared is through the timing of the use of a distributed encryption system, which was developed by the authors.

Erich Rice, Dennis Guster & Richard Anderson
Information Systems
St. Cloud State University
St. Cloud, MN 56301
eprice@stcloudstate.edu
dcguster@stcloudstate.edu
richardalmanderson@gmail.com

Introduction

The Internet is a very hostile place and because public clouds are connected to the Internet the threat permeates them as well. There have been numerous instances of successful attacks by hackers as illustrated by the successful attacks on the likes of Equifax, Anthem and the Democratic National Committee among many others. The literature also indicates that if properly managed and configured, the new forms of the encryption algorithms can be a quite effective tool at thwarting these attacks (Abdalla, M., Bellare, M., & Neven, G., 2010). The literature also indicates that too often the algorithms can be misused, a good example is with the protocol secure shell (ssh). In that a key is created and stored for long periods of time and it gets compromised while stored on the client-side host device (Ylonen, T. & Lonvick, C., 2006).

One effective way of dealing with the key compromise strategy is to make the key more complex. This can be done with a key agreement protocol such as is described in (Ylonen, T. & Lonvick, C., 2006). In this case both the client and the server would need to be compromised (Filipe, J. & Obaidat, M., 2008). Also, there are various scenarios where a key management server, such as in a Public Key Infrastructure (PKI) setup is used and, in such cases, that server would need to be compromised to obtain the key (Mearian, L., 2009). However, the methodology utilized herein involves splitting the key and storing it among multiple hosts (Ylonen, T. & Lonvick, C., 2006). This method offers the advantages of fault tolerance and obscurity because padding can be used on each segment. Further, the fault tolerance is obtained by placing the sub-keys on more than the needed amounts. For example, the sub-keys could be placed on five hosts with an N-2 algorithm. In such a case any two of the five nodes could fail and the total key could still be recovers. For a detailed explanation of this process see (Redman, J., et al, 2013).

This process works quite well within LAN based private cloud where the latency is small. However, implementing such a strategy is definitely more challenging in a public cloud which requires that access be obtained via the WAN-based Internet. This was especially true in our experiment in which the public cloud was 14 hops away from the client using the keys to authenticate. The literature certainly does indicate that some application does not run well across a WAN, particularly in cases of large data transfers and dependence on sliding window protocols (Fall, K., & McCanne, S. 2005). The challenge then is to ascertain if the latency problem is a function of the network or the application itself (Guo, C. et al. 2015). Of course there could be an interaction between the network and the application which would further complicate the matter. The literature does indicate that latency can be a problem in key management systems (Nash, R., 2016). The latency problem as related to key transfer has been addressed and new protocols designed to combat the problem. A good example is (Zhang, X., Lam, S., Lee, D-Y., & Yang, Y. 2003) which uses a multi-cast strategy to make sure the key arrives in a timely manner and the inquiry does not get lost. While this research is useful in LAN-based applications, little research has been done that addresses the application of a distributed key scheme in a public-based Cloud. To gain an understanding of the latency metric in such a scenario this paper will focus on measuring the latency of authentication using a distributed key algorithm. Specifically, several different algorithms will be tested in two different public Clouds and the results analyzed to ascertain differences between the two Clouds. Further, the data will be examined to ascertain the probability of any trial exceeding the three second target threshold.

Literature Review

Distributing parts of the encryption algorithm across multiple nodes is not a completely new concept. However, its use in public Cloud enterprise level applications as suggested herein has not been widely done. The basic concept of distributed encryption was suggested and implemented in the mid-1990s. An example of such an implementation is offered in (Postma, A., De Boer, W., Helme, A., & Smit, G., 1996). This work suggests the n-t algorithm in which n = number of nodes and t = the number of nodes that could be faulty and the key could still be recovered. In this system the encryption scenario is started at a root node and follows a preset path (or redundant path) until all the needed parts of the key are recovered.

This basic logic has been refined by treating each one of the segments as an independent entity, but still utilizing the n-t logic. The definitive work in this regard (Guster, D., Brown, C., & Sultanov, R. 2010) used relatively small keys (16 bytes) to implement a proof of concept scenario. Further, while the key was 16 bytes only 12 of the bytes were significant and a sliding window pattern was used so that even with n = 4 the key could still be recovered even if 2 of the 4 nodes was down.

The basic ideas were further refined by (Rego, I., Gillespie, N., & Guster, D., 2011) in which a UNIX script provided the logic to split an RSA algorithm across several hosts. In this case a pre-generated key would be read from a file and the output from the UNIX script would be considered as fractions of the initial file. When the fragments are re-formed, they first would be checked using a file hash such as MD5sum. While this is a new twist in the form presented it lacked the complexity required for a production system.

The idea has been refined to the point it could be deployed to a production system by (Redman, J., et al, 2013). This algorithm has been expanded to use 24 byte keys, AES-192 encryption and up to 6 nodes. This will be the algorithm that is used herein to test the performance of the AWS and DigitalOcean Clouds.

Latency has been a problem in distributed applications for some time (Ramaswamy, R., 2000). A detailed treatment of the typical problems coupled with expected metrics can be found in (Brook, A., 2015). This work also discusses how the concept of threading can be used to achieve encapsulation as well as speed up a given application which is the main concept used herein. Some of the solutions that are possible by upgrading a datacenter to low-latency ethernet are also discussed in (Rumble, S., Ongaro, D., Stutsman, R. Rosenblum, M. & Ousterhout, J., 2011). However, unless private Clouds and the Internet as a whole adopt such technology it will have very limited influence on the problem discussed herein.

A key management system offers a special challenge in regard to latency, in that if the response is not adequate and retries occur it delays the use of any application associated with the target host. The basic latency challenges related to a key management system have been described by

(Nash, R., 2016). There have been protocols designed to combat latency issues such as (Zhang, X., Lam, S., Lee, D-Y., & Yang, Y. 2003), but such methodologies that use a multi-cast strategy may not be effective in a WAN inquiry mode because they rely on resending packets to known hosts. Work has been done related to describing and combating latency problems with public Clouds (Xu, Y., Musgrave, Z., Noble, B. & Baily, M., 2013) which indicates that for some applications latency can indeed be a major problem. There does not appear to be published research that directly deal with latency issues with distributed fault tolerant keys. Therefore, the experimental trials described herein can be useful in determining the effectiveness of such strategies and guide future generations of such systems.

Methodology/Results

Randomness in the Key: It is critical to generate the key using an effective random number generator (Eastlake, D., Crocker, S., & Schiller, J., 1994). The experimentation from (Redman, J., et al, 2013) confirmed that the quantum random number generator did offer better results and it was used to generate the keys for the current tests as well.

Splitting the Key: There are numerous methods that can be used to split a key into subparts. However, because the goal in this study was to ensure both encapsulation and reliability an N-3 algorithm was tested. For this series of trials, the key was broken up into 24-byte segments, with each byte being made up of 8 bits, which can be represented by two hexadecimal characters. For these trials the 24-byte / 192-bit key size was utilized. To help ensure secrecy the AES (Advanced Encryption Standard) encryption was chosen due to its recognized robustness.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
D8	77	9B	57	4A	BB	43	34	5F	77	96	C3	42	19	42	79	47	2A	D6	E5	A1	B3	50	4F

IV= 99BA8C95FBB459DBE5E92C769FFC55BB
SALT=2B336181CE15142C

					BB	43	34	5F	77	96	C3	42	19	42		47	2A	D6	E5	A1		50	
	77	9B	57	4A					77	96	C3	42	19	42			2A	D6	E5		B3	50	4F
D8		9B	57	4A		43	34	5F				42	19	42	79			D6	E5	A1	B3		4F
D8	77		57	4A	BB		34	5F		96	C3			42	79	47			E5	A1		50	4F
D8	77	9B		4A	BB	43		5F	77		C3		19		79	47	2A				B3	50	4F
D8	77	9B	57		BB	43	34		77	96		42			79	47	2A	D6		A1	B3		

Figure 1: Key (Top Boxes) with Initialization Vector (IV) and Salt (Redman, J., et al, 2013)

Figure 1 above displays a sample 24-byte key in hexadecimal which will require 48 hex characters. Below the key is a sample table of how the sample key might be split up. Specifically, the six rows in the bottom part of the table represent the six nodes contained in the distributed key system and depict how the key might be broken up using the method described herein. The blank spaces in the table above represent the missing pieces of the key. Note that they are strategically placed so that the whole key can be recovered if any three of the sub-keys are recoverable.

The Distributed Key Environment: A distributed encryption environment was created on each of the two public Clouds with each Cloud receiving the same configuration. Specifically, the various nodes utilized a virtual machine architecture using the Xen Server logic and all the servers ran Node.js, which utilizes JavaScript which facilitates "fast, scalable" deployment of network applications. Node.js was chosen because it uses an event-driven non-blocking I/O model. This logic makes it lightweight and efficient which is perfect for data-intensive real-time applications that run across distributed devices (Nodejs.org, 2019). In terms of an operating system, all of the nodes ran Ubuntu Linux 12.04 64-bit Edition.

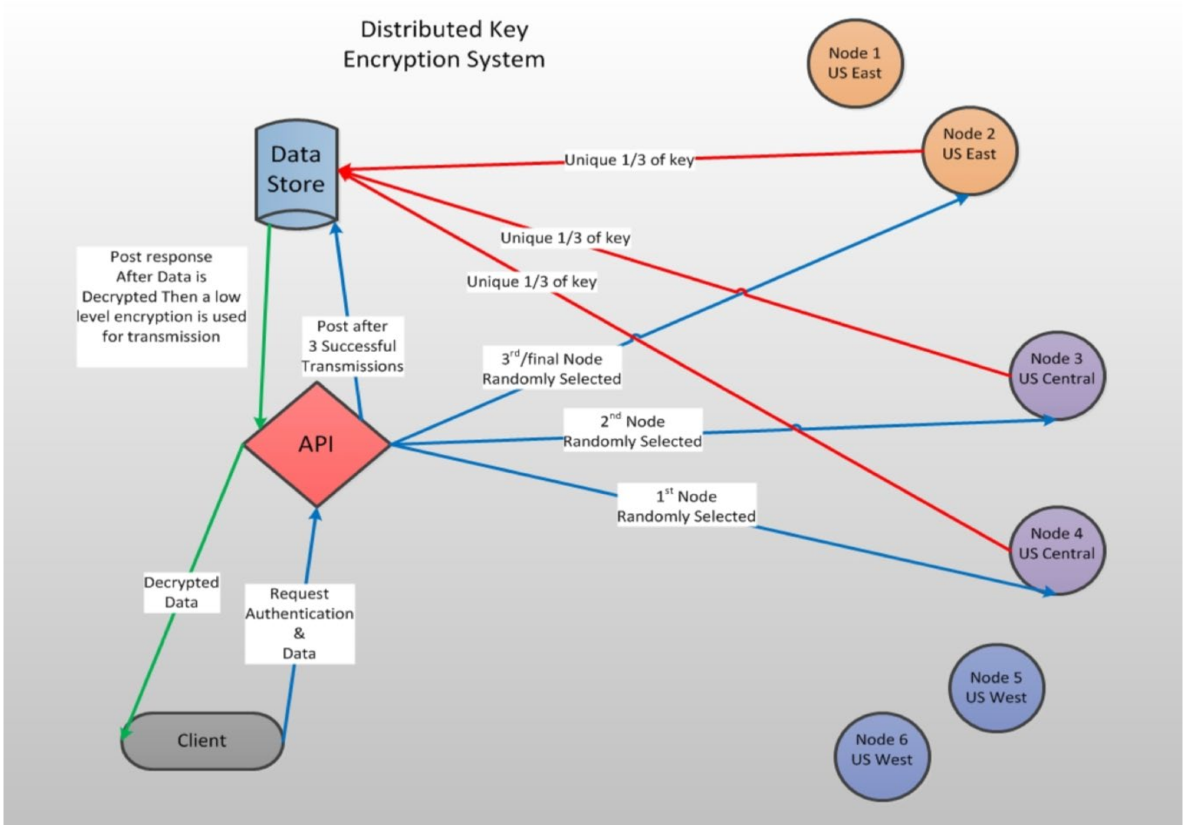


Figure 2: Distributed Encryption System (Redman, J. et al, 2013)

Figure 2 above provides a visual depiction of how the distributed keys would be accessed and assembled to form the full key. Note that an N-3 algorithm is being used in this example. In the experiment herein all nodes would reside within a single public Cloud region, i.e. N. Virginia.

The Results of the Trials: The measures of central tendency for each of the two public Clouds tested: Amazon Web Service and DigitalOcean are depicted in the two Tables below. The Table 1 includes five different algorithms and as expected, as the number of nodes increases so does the delay. Implementing some type of fault tolerance (t < n) also generally adds to the delay.

Test Type	3 of 6 Nodes	4 of 8 Nodes	5 of 8 Nodes	6 of 8 Nodes	8 of 8 Nodes
Min	287 ms	297 ms	503 ms	516 ms	489 ms
Max	1516 ms	1559 ms	1802 ms	1669 ms	1574 ms
Average	570.68 ms	589.41 ms	601.30 ms	605.23 ms	612.56 ms
Standard Deviation	132.96 ms	133.99 ms	157.01 ms	96.53 ms	134.05 ms

Table 1: Amazon Web Service Cloud (AWS)

Test Type	3 of 6 Nodes	4 of 8 Nodes	5 of 8 Nodes	6 of 8 Nodes	8 of 8 Nodes
Min	292 ms	297 ms	477 ms	491 ms	486 ms
Max	1666 ms	1533 ms	1836 ms	2030 ms	1407 ms
Average	520.96 ms	522.71 ms	528.55 ms	539.05 ms	547.20 ms
Standard Deviation	64.06 ms	27.99 ms	42.85 ms	40.33 ms	41.66 ms

Table 2: DigitalOcean Cloud (DO)

On the surface even if the max values were used to assess latency it would appear that the 3 second target was met, but that value is just really part of the equation. Given the data above even the 292ms value could be problematic in meeting the 3 second target. While ~.292 of a second may not seem that significant, one needs to look at the total response time model which is quite complex and involves a number of components. So, the real test then becomes end-to-end delay. A good representative example of this model is offered by (Fleming, D., 2004).: **User - Application-Command _CPULocalComputer _NICLocalComputer _Network-Propagation _Switch _Network-Propagation _Switch _Network-Propagation _NICFile-Server _CPUFile-Server _SCSI-bus _DiskRead**, (…then traverse the path in reverse for the reply).

Given that the network delay on the Internet in the US might often take .5 seconds in each direction it is important to optimize each of the parameters. Further, one needs to realize that this whole algorithm is based on queuing theory, which means that there is an interaction among all the parameters. In other words, a delay of .292 instead of .1 at the first parameter won't simply result in .192 seconds of additional response time. Rather, it will propagate through the entire algorithm and the delay will get a little longer with added wait time at each successive parameter. To put this in perspective, if one assumes a geometric progression through all 12 parameters in the algorithm above the result in total added delay would probably exceed 1 second.

Tables 3 and 4 below provide more detailed information about the effect of implementing fault tolerance and having to deal with down nodes which may require additional inquires to more than just the minimum number of nodes. It is clear that this scenario definitely increases latency in some cases almost to the two second delay level. These values would be troubling and more than likely when figuring end-to-end delay would exceed the 3 second target threshold. These results indicate that the basic model could be employed and meet the target total end-to-end delay target of 3 seconds or less but as the models become more complex the probability of meeting the target lessens. One of the primary questions of this study is a comparison of the two public Clouds and a quick look at the raw data indicates similar results. However, Figures 3 and 4 are provided below that depicts the latency relationship in more detail.

Test Type	**3 of 6 Nodes (3 Nodes Down)**	**4 of 8 Nodes (4 Nodes Down)**	**3 of 8 Nodes (5 Nodes Down)**
Min	518 ms	511 ms	523 ms
Max	1626 ms	1879 ms	1749 ms
Average	641.40 ms	672.81 ms	696.75 ms
Standard Deviation	122.09 ms	131.79 ms	156.39 ms

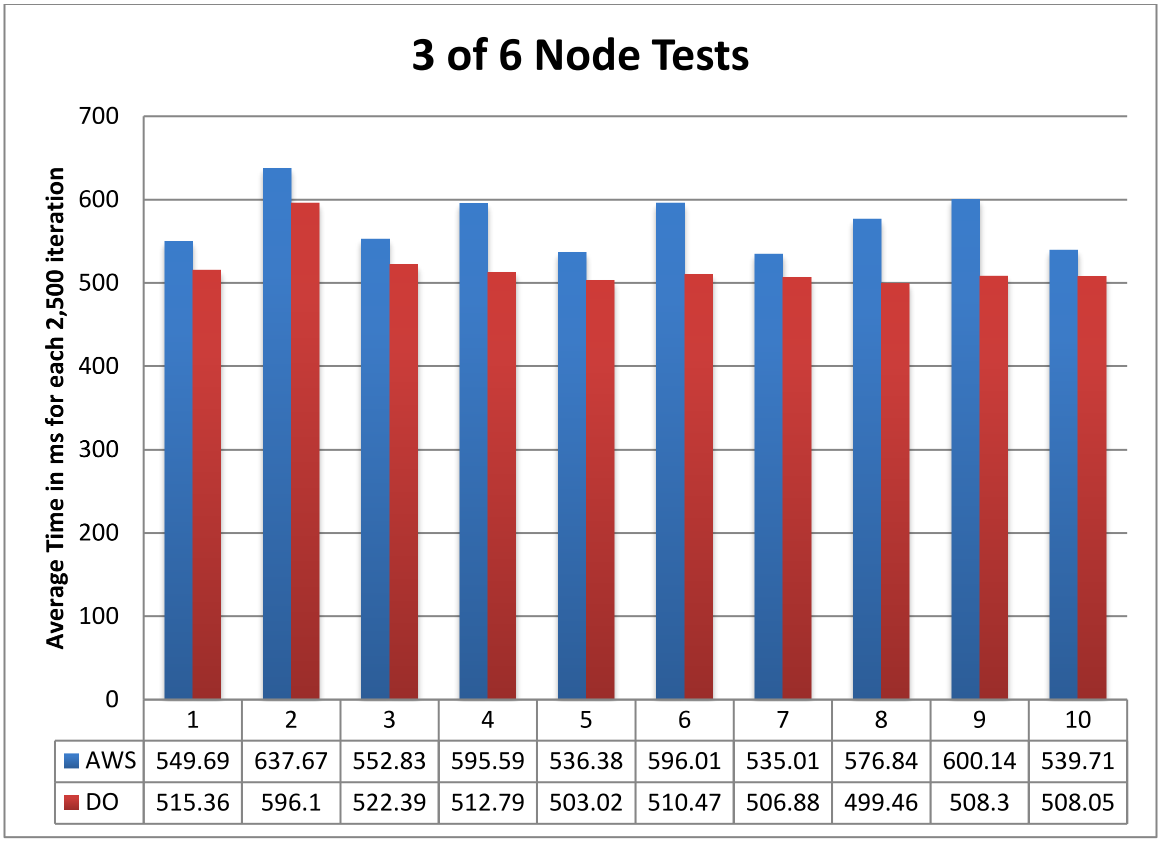
Table 3: Timing Results AWS (Nodes Down)

Test Type	**3 of 6 Nodes (3 Nodes Down)**	**4 of 8 Nodes (4 Nodes Down)**	**3 of 8 Nodes (5 Nodes Down)**

Min	495 ms	495 ms	507 ms
Max	2378 ms	2712 ms	1890 ms
Average	569.50 ms	594.04 ms	612.45 ms
Standard Deviation	62.20 ms	56.77 ms	49.38 ms

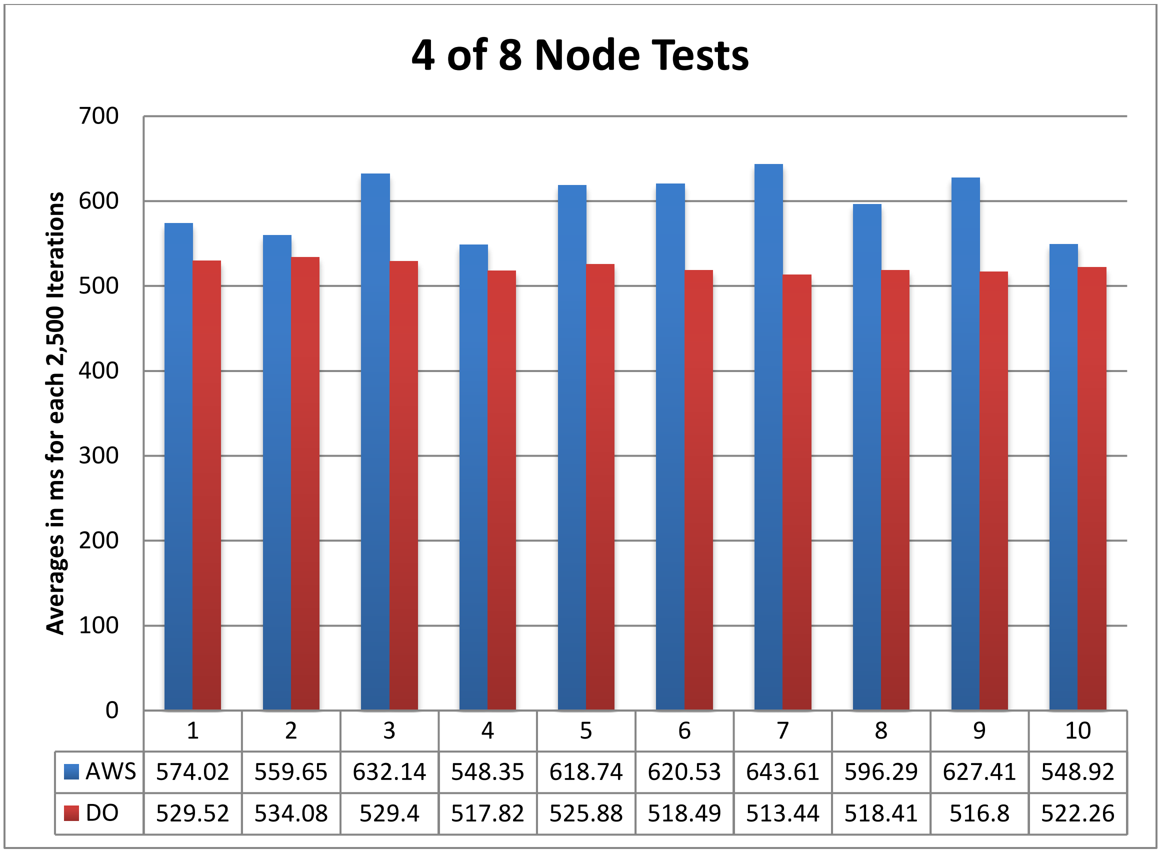
Table 4: Timing Results DO (Nodes Down)

Comparison of the Two Public Clouds: There was definitely an advantage in reduced latency in favor of DigitalOcean. In Figure 3 (3 of 6 nodes) in which no nodes are down the difference is fairly consist with about 25 ms or about 5% difference. Further, enough data points were collected to be fairly comfortable with these values from a sampling perspective. While the difference does not seem significant statistically when one calculates end-to-end delay as described above one might expect an added 50 ms if a geometric distribution holds. Figure 4 (4 of 8 nodes) with no nodes down exhibited larger differences typically in the 75 ms range or about a 12% difference. This would be expected because this algorithm is more complex than the 3 of 6 node method. Once again this would have a more detrimental effect on end-to-end delay perhaps as much as 150 ms. This difference while a concern would not make the 3 second target implausible.



	1	2	3	4	5	6	7	8	9	10
🟦 AWS	549.69	637.67	552.83	595.59	536.38	596.01	535.01	576.84	600.14	539.71
🟥 DO	515.36	596.1	522.39	512.79	503.02	510.47	506.88	499.46	508.3	508.05

Figure 3: Averages for 3 of 6 Tests Amazon Web Service (Blue) and Digital Ocean (Red)



	1	2	3	4	5	6	7	8	9	10
🟦 AWS	574.02	559.65	632.14	548.35	618.74	620.53	643.61	596.29	627.41	548.92
🟥 DO	529.52	534.08	529.4	517.82	525.88	518.49	513.44	518.41	516.8	522.26

Figure 4: Averages for 4 of 8 Tests Amazon Web Service (Blue) and Digital Ocean (Red)

Discussion/Conclusions

In general, the results were encouraging. The basic algorithms were proven to operate on average in an acceptable latency range and while the more complex algorithms added to the delay they were still in the acceptable range. However, some of the max values observed reached the 1.5 second ranges with no nodes down and exceeded 2 seconds with nodes down. In those particular cases one would expect the end-to-end delay to exceed the 3 second threshold.

While cloud computing is the prevailing computing architecture it presents numerous challenges due to the complexity of its infrastructure. It shines when the major computing takes place within the cloud and the hardware supporting the virtual machines is not over subscribed. However, getting to the Cloud and receive the results of a query can also add to the overall delay. A couple of quick tests from the authors private Cloud to the two public Clouds revealed that the initial route to DigitalOcean was 13 hops which equated to a delay of 49 ms. Whereas, the route to Amazon Web Service was 18 hops or 86 ms. However, the delay over time reached the 300 ms range. In contrast, the DigitalOcean delay remained in the upper 40 ms range. So, based on the geographic location of the authors DigitalOcean (San Francisco, CA based datacenter) provided an advantage over Amazon Web Service's (Seattle, WA based datacenter) based on geography and the manner in which the Internet is implemented.

With compromising the key being one of the strategies of choice for hackers it is of utmost importance to protect the key. This study showed that distributing the key in a fault tolerant manner within a public Cloud was indeed practical. Further, there have been cases in which a hacker tried to compromise the key and was not successful but inadvertently (or intentionally) deleted the key. This would be a most detrimental situation because the company/individual in question would not be able to decrypt and recover their data. In the case of the algorithms used herein the hacker would need to compromise several hosts and ascertain how to assemble the key from the subparts obtained from the multiple hosts utilized to actually access the data protected by the encryption.

In summary, dealing with latency issues within a public Cloud still merits investigation. Tuning applications within public Clouds can be a difficult undertaking because the actual control of the infrastructure is vested with the cloud provider and the range of observed latency probably will be wider and more erratic then if the same application was placed on a private Cloud. The same algorithms when tested on the authors' private Cloud typically ran in the 100 ms range, a significant improvement. Given that not everyone can be expected to maintain their own private Cloud and given the cost advantages that public Clouds provide one can expect more and more applications to be migrated to these public Cloud providers such as the AWS and DigitalOcean. One design aspect that needs to be carefully considered is latency and particularly the latency over time particularly during high traffic times, such as for a retail business during the holiday shopping season. While the primary goal of this study was to test the distributed key algorithm in a public Cloud environment the same basic logic could be most useful in determining the viability of any application to function effectively within a public Cloud.

REFERENCES

Abdalla, M., Bellare, M., & Neven, G. (2010). Robust Encryption. Cryptology ePrint Archive, Report 2008/440 Retrieved from http://eprint.iacr.org/2008/440.pdf.

Brook, A. (2015). Evolution and Practice: Low Latency Distributed Application in Finance. ACM Que: Distributed Computing, 13:4.

Eastlake, D., Crocker, S., & Schiller, J. (1994). Randomness Recommendations for Security. IETF RFC 1750, December 1994, Retrieved from http://www.ietf.org/rfc/rfc1750.txt.

Fall, K., & McCanne, S. (2005). You Don't Know Jack about Network Performance. ACMQUE: Networks, 3(4).

Filipe, J. & Obaidat, M. (2008). Two Types of Key-Compromise Impersonation Attacks against One-Pass Key Establishment Protocols, Communications in Computer and Information Science, 2008/23, p. 227-238.

Fleming, D. (2004). Network Response Time for Efficient Interactive Use. Proceedings of the 20th Computer Science Seminar, Addendum-T2-1. RIP, Hartford Campus, April, 24, 2004.

Guo, C., Yuan, L., Xiang, D, Dang, Y., Huang, R., Maltz, D., … Kurien, V. (2015). Pingmesh: A Large-scale System for Data Center Network Latency Measurement and Analysis. A paper presented at ACM SIGCOMM, London, UK.

Guster, D., Brown, C., & Sultanov, R. (2010). Enhancing Key Confidentiality by Distributing Portions of the Key Across Multiple Hosts. Retrieved from http://micsymposium.org/mics_2010_proceedings/paper_11.pdf

Mearian, L. (2009). Sun offers open source encryption key management protocol. Retrieved from http://www.computerworld.com/s/article/9128101/Sun_offers_open_source_encryption_ key_management_protocol.

Nash, R. (2016). Key Management Challenges and Best Practices. http://searchfinancialsecurity.techtarget.com/tip/Key-management-challenges-and-best-practices.

Postma, A., De Boer, W., Helme, A., & Smit, G. (1996). Distributed Encryption and Decryption Algorithms. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.3186.

Ramaswamy, R. (2000). Latency in distributed, sequential application designs, ACM SIGSOFT Software Engineering Notes, v.25 n.2, p.51-55, March 2000 [doi>10.1145/346057.346072]

Redman, J., Rice, E., Han, S., Anderson, R., Paulson, B., and Schwarting, J. (2013). Can A Distributed Key System Broken Up Over Multiple Nodes Provide Greater Security Robustness While Meeting System Performance Requirements. Retrieved from http://www.micsymposium.org.

Rego, I., Gillespie, N., & Guster, D. (2011). Implementing a Distributed Key Algorithm to Enhance Confidentiality and Provide Fault Tolerance. Retrieved from http://www2.css.edu/mics/Submissions/submissions/Implementing%20a%20Distributed%20Key%20Algorithm%20to%20Enhance%20Confidentiality%20and%20Provide%20Fault%20Tolerance.pdf.

Rumble, S., Ongaro, D., Stutsman, R. Rosenblum, M. & Ousterhout, J. (2011). It's time for low latency, Proceedings of the 13th USENIX conference on Hot topics in operating systems, May 09-11, 2011, Napa, California.

Xu, Y., Musgrave, Z., Noble, B. and M. Baily. (2013) Bobtail: Avoiding Long Tails in the Cloud. A paper presented at the 10th USENIX Symposium on Networked System Design and Implementation (NSDI) 329-341.

Ylonen, T. & Lonvick, C. (2006). The Secure Shell (SSH) Authentication Protocol, IETF RFC 4252, Retrieved from http://tools.ietf.org/html/rfc4252.

Zhang, X., Lam, S., Lee, D-Y., & Yang, Y. (2003). Protocol Design for Scalable and Reliable Group Rekeying. IEEE/ACM Transactions on Networking. 11(6). 908-922.