RESEARCH ARTICLE

SAMRIDDHI Volume 15, Issue 1, 2023 Print ISSN: 2229-7111 Online ISSN: 2454-5767

Impact of TLS Overhead on Segmented Network for Cloud Database Systems

Suresh P. Kannojia*, Jitendra Kurmi

Department of Computer Science, University of Lucknow, Lucknow, Uttar Pradesh, India

Abstract

Cloud database serves flexible, affordable, and scalable database systems. Even the cloud database is secure with transport layer security (TLS), but the performance overhead that TLS introduces when executing operations on one of the major No SQL databases: Mongo DB in terms of latency. To explore TLS performance overhead for Mongo DB, we performed two tests simulating common database usage patterns. We first investigated connection pooling, where an application uses a single connection for many database operations. Then, we considered one request per connection in which an application opens a connection, executes a process, and immediately closes the connection after completing the operation. Our experimental result shows that applications that cannot endure significant performance overhead should be deployed within a properly segmented network rather than enabling TLS. Applications using TLS should use a connection pool rather than a connection-per-request.

Keywords: SSL/TLS, Mongo DB, Cloud, Segmented network, Performance.

SAMRIDDHI : A Journal of Physical Sciences, Engineering and Technology (2023); DOI: 10.18090/samriddhi.v15i01.22

Introduction

Advances in cloud computing have created the need to store and manage large amounts of unstructured data in distributed databases while providing high availability and scalability, therefore, No SQL databases is used to fill this gap. These databases are being increasingly used to store sensitive information. In this cutting-edge, security is becoming a higher priority for organizations using such databases. Common attacks on database systems include eavesdropping i.e an attacker reads the communication with the database. On the other hand, an attacker spies on the transmission and alters the information i.e., man-in-the-middle (MITM) attacks.

Moreover, an inside threat should not be overlooked, making it necessary to protect internal network traffic. In general, these integrity and confidentiality problems are mitigated through encryption. A common way to provide encryption on network communication is through SSL/TLS. This prevents attackers from intercepting sensitive data transferred between the application and the No SQL server. Unfortunately, enabling this feature reduces the communication performance with the No SQL database server. Quantifying this performance degradation allows businesses to decide whether the security benefit is worth the additional cost. This paper describes the precise performance overhead that TLS introduces when executing operations on Mongo DB No SQL database server. To see the performance overhead, we planned to conduct two tests simulating common database usage patterns on Mongo DB:

Corresponding Author: Suresh P. Kannojia, Department of Computer Science, University of Lucknow, Lucknow, Uttar Pradesh, India, e-mail: spkannojia@gmail.com

How to cite this article: Kannojia, S.P., Kurmi, J. (2023). Impact of TLS Overhead on Segmented Network for Cloud Database Systems. *SAMRIDDHI : A Journal of Physical Sciences, Engineering and Technology,* 15(1), 171-175.

Source of support: Nil

Conflict of interest: None

Test 1: Investigating connection pooling where an application uses a single connection for many database operations. This minimizes the TLS overhead introduced by opening a new link.

Test 2: We considered one request per connection in which an application opens a connection, executes an operation, and immediately closes the link once the process is complete. This introduces more performance overhead when compared to a connection pool.

Literature Review

The performance, availability, and elastic scalability are prioritized in cloud storage services with No SQL systems such as MongoDB, DynamoDB, Cassandra, HBase, or Voldemort. While the distribution strategy may differ, all of these systems have one thing in common: they usually forego enterprise-grade capabilities in return for improved performance or availability. A classic example of a weak consistency model

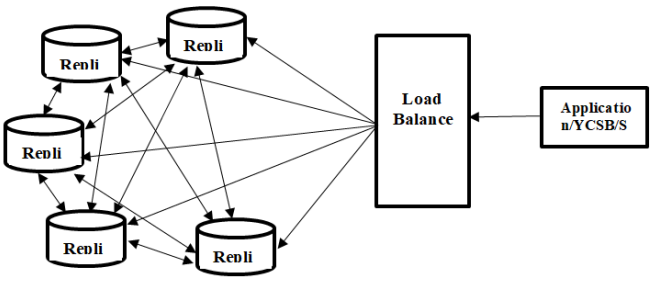

Figure1: Overview of methodology for Cloud Database System

is given by D. Bermbach and J. Kuhlenkamp,[1] which is based on a comparison between CAP theorem and PACELC model.[2-3] An eventual consistency model is proposed to achieve high availability for distributed computing and has no security mechanism as an optional feature presented by W. Vogels.[4] The performance overhead of SSL for client-server communication was explored by Apostolopoulos *et al.*,[5] Kant *et al.*,[6] Zhao *et al.*,[7] and Coarfa *et al.*[8] Here, the outcome of this extremely simple situation is significantly more predictable than the cloud database system because just two machines are involved. Shirasuna *et al.* examine the impact of TLS on SOAP-based web services performance.[9] Cloud database systems cannot utilize the measuring approach for more comprehensive benchmarking because they uses a simple echo service. Various research studies investigated the performance overhead of TLS in cloud database systems while using obsolete protocols by using Session Initiation Protocol (SIP) or Internet Protocol (IP) in the ISO/OSI hierarchy.[10-12]

On the other hand, existing methodologies do not consider any security mechanisms in their research for benchmarking cloud database systems.[13-18] In literature, a dedicated benchmarking tool for a cloud database system to ensure service quality was proposed by M. Grambow et.al.[19] A pattern-based approach reduces the efforts for defining micro-services benchmarks for cloud databases.[20] A comparative analysis of various TLS libraries which includes authenticated encryption cipher, hashing, and public-key cryptography but does not cover the cloud system.[21] Various TLS libraries have been analyzed using the cryptographic token interface, CPU-assisted cryptography with AES-NI, thread safety, and language support.[22] None of the researchers has discussed the TLS overhead with No SQL database like Mongo DB. So it motivates us to use Open Source Yahoo! Cloud Serving Benchmark tool (YCSB), to measure the performance overhead of TLS for Mongo DB database. The impacts of TLS overhead over cloud play a significant role on data security in cutting-edge communication.

Methodology

Communication in cloud computing depends on three primary responsibilities in cloud database systems include:

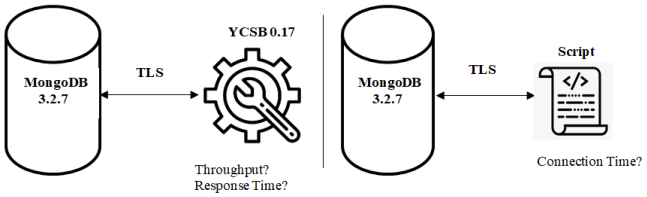

Figure 2: Overview of connection pool tests.

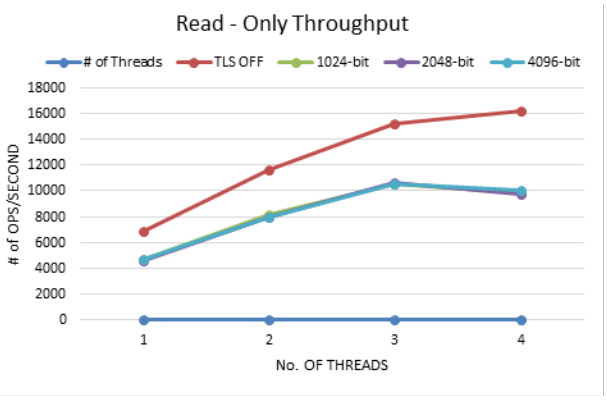

Figure 3: TLS Read-only throughput Overhead on Mongo DB

Step 1 To replicate the stored data across multiple machines as per user requirement

Step 2 Load Balancer used to maintain the load of the cloud database system and proxy interface for client.

Step 3 Application provides an interface to user for the exchange of data.

The overview of the methodology is shown in Figure 1.

The implementation consist of two phase to measure the performance overhead of TLS with respect to throughput, response time and connection time.

Phase 1: With Enabled TLS

We ran the Yahoo! Cloud System Benchmark (YCSB) 0.17.0 with Java 8 update 92 on a Mongo DB 4.0 instance with TLS enabled when conducting the connection pool tests. We modified the Mongo DB client used by the benchmark suite to use TLS. When performing the one request per connection

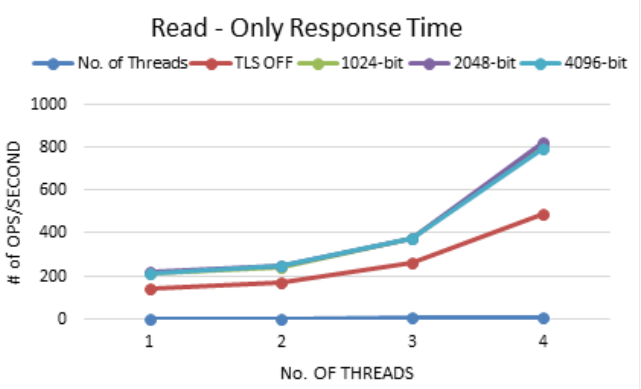

Fig. 4: TLS Read-only response time throughput overhead on Mongo DB

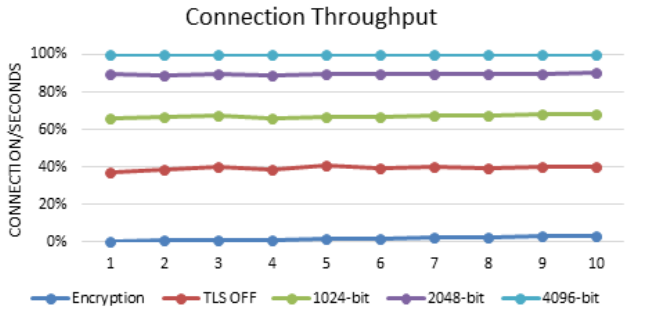

Figure 5: TLS Connection throughput Overhead on Mongo DB

tests, we created a custom script using Node.js version 17.0. We enabled TLS on Mongo DB by following the steps outlined in the Mongo DB documentation.

Using TLS v1.3, although we varied the key length for the initial RSA handshake, the symmetric cipher used for communication remained the same for all tests (namely AES256-GCM-SHA384). TLS generated the server-side certificate before the start of testing. TLS could argue that using an RSA handshake isn't an optimal approach to TLS, from both a security and performance perspective shown in Figure 2.

Phase 2: without enabled TLS

We have performed the same experiment as above (phase1) except the enabled TLS.

Therefore, we looked into alternatives like the Diffie-Hellman key exchange or its elliptic curve variant. But we didn't use these ciphers in this paper. We decided to maintain the original setup for our tests to resemble a realistic scenario.

Experimental Setup

We performed connection pool and one request per connection tests on an Intel Core i3-3217U 1.80GHz CPU (4 cores) with 8GB of RAM, Samsung HDD 500GB storage drive running Windows 10 Pro, having Mongo DB version 4.0 and testing tools.

Result & Analysis

Test 1: Connection pool

For this test, YCSB opens a connection to Mongo DB, executes a test suite through the same link, and closes the connections upon completion. Since we were only interested in the overhead introduced by TLS, it was sufficient to run only read operations rather than a combination of update and read procedures. To run the test, we used the following script with 1024, 2048, or 4096-bit key size.

YCSB Tool Script

```
1. #!/bin/bash
2. mongo ycsb --eval 'db.dropDatabase()' --tlsCAFile public.
crt --tls --host localhost
3. bin/ycsb load mongodb -s -P workloads/workloadc
-p recordcount=20000 -p mongodb. url=mongodb://
localhost:28013 > tls_load_c.txt
4. for threads in 1 2 4 8 ;
5. do
6. bin/ycsb run mongodb -s -threads ${threads} -P
workloads/workloadc -p operationcount=20000000
-p mongodb.url=mongodb://localhost:28013 > tls_
run_c_${threads}.txt
7. done
```

Initially, the script drops the database, ensuring that we started with an empty database (line 2). Next, YCSB creates the database and loads 20,000 records using the C test suite (workload c) from the YCSB benchmark (line 3). Subsequently, YCSB performs 20 million read operations with the C test suite using 1, 2, 4, and 8 threads, respectively (lines 4-7). Each thread connects separately to the database, and together they complete all operations. We ran this script with different TLS server certificates to test the cases mentioned above: no TLS and TLS with 1024, 2048, and 4096-bit key size. We didn't use the URL parameter 'ssl=true' to enable TLS on a Mongo DB connection since we made small changes to YCSB that enable TLS connections upon setup. When we examined the throughput, measured in the number of operations per second, we noticed that enabling TLS introduces an overhead between 29% and 40%, shown in Table 1 and Figure 3.

Since the tests ran on a quad-core machine, the machine expected the slight setback of using eight threads. We can also conclude that the difference in speed using different key lengths is negligible. This was also expected due to the overhead created by the key size that only occurs at the time of connection, the majority of time is spent on executing the operations themselves. Note that the test only creates as many connections as per threads.

The observed response time overhead of TLS from the same test shown in Table 2 and Figure 4.

Unfortunately, we weren't able to compute a valid standard deviation. This was because YCSB provides raw data in milliseconds. However, most operations and their averages (provided in microseconds) were under one millisecond.

Test 2: Connection time

```
1. var express = require('express');
2. var mongodb = require('mongodb');
3. var async = require('async');
4. var fs = require('fs');
5. var app = express();
6. var MongoClient = mongodb.MongoClient;
7. var url = 'mongodb://localhost:28013/?ssl=true';
8. var certFileBuf = fs.readFileSync('mongodb-cert.crt');
9. var options = {
10. server: {tlsCA: certFileBuf}};
12. app.get('/run', function (req, res) {
13. var times = [];
14. async.timesSeries(10, function (n, callback) {
15. ar start = new Date().getTime();
16. async.timesSeries(100, function (n, callback) {
```

Table 1: Comparison of TLS Read-only throughput overhead between No. of threads and Key Size

No. of Threads	*1*		*2*		*4*		*8*	
Key Size	No. of Ops/ Second	Percentage (%)	No. of Ops/ Second	Percentage (%)	No. of Ops/ Second	Percentage (%)	No. of Ops/ Second	Percentage (%)
TLS OFF	6871.50	0.00	11598.24	0.00	15201.50	0.00	16151.18	0.00
1024-bit	4724.95	31.24	8185.46	29.42	10579.21	30.41	9877.67	38.84
2048-bit	4609.37	32.92	7982.27	31.18	10656.64	31.18	9696.51	39.96
4096-bit	4714.12	31.40	7978.18	31.21	10575.03	30.43	10030.70	37.89

Table 2: Comparison of TLS Read-only response time throughput overhead between No. of threads and Key Size

No. of Threads	*1*		*2*		*4*		*8*	
Key Size	No. of Ops/ Second	Percentage	No. of Ops/ Second	Percentage	No. of Ops/ Second	Percentage	No. of Ops/ Second	Percentage
TLS OFF	143.59	0.00	170.16	0.00	260.35	0.00	490.17	0.00
1024-bit	209.6	45.98	241.96	42.20	374.98	44.03	804.00	64.02
2048-bit	214.95	49.70	248.16	45.84	372.40	43.04	818.98	67.08
4096-bit	210.14	46.35	248.38	45.97	375.19	44.01	791.87	61.55

Table 3: Comparison of TLS Connection throughput overhead between Average connection/second and Key Size

Encryption	*Average connections per second*	*Percentage (%)*	*Standard Deviation*
TLS OFF	105.4702	0.00	5.62231
1024-bit	76.91441	27.07	3.546171
2048-bit	62.55387	40.96	2.065384
4096-bit	28.78176	72.71	0.80246
Average		46.91	

```
17. MongoClient.connect(url, options, function (err, db) {
18. db.close();
19. callback();});}, function () {
22. var end = new Date().getTime();
23. times.push((end - start).toString());
24. callback();});}, function () { res.send(times);});});
30. var server = app.listen(3000, function () {
31. console.log('listening on port %d', server.address().
port);});
```

We expected that most performance overhead is introduced while making a new connection despite the performance overhead of a connection pool (as measured in test #1). This test examined the theory by measuring connection time using the following script:

The script opens (line 19) and closes (line 20) a connection to Mongo DB a hundred times (line 18) while recording the whole time (lines 17 and 24–25). It repeats this test 10 times, taking the average of the consecutive runs (line 16).

When extracting the raw data, we can see the average connection per second and standard deviation of the ten consecutive runs. It's important to note an overhead moving from 27.07% for 1024-bit key size, 40.96% for 2048-bit key size and 72.71% for 4096-bit key size concerning a connection without TLS. Thus, 4096-bit TLS connections are up to four times slower to establish than unencrypted connections shown in Table 3. The average connection overhead in terms of latency is 46.91 %.

The following graph illustrates the elapsed time for each run and presents it as connections per second shown in Figure 5.

Conclusions and Future Scope

In this paper comparative analysis with experimental result had been performed, we found that TLS overhead is high, especially for applications using one connection per request with 4096-bit key size. The difference between establishing a connection for different key lengths is negligible for an application that maintains long-running connections. There is still a large overall overhead compared to unencrypted connections (around 46.91% in latency). Therefore, enabling TLS on a MongoDB database increases the throughput performance overhead between 27.07% - 40.96%. If such a performance hit is not an option, TLS cannot use to protect from MITM attacks on the internal network. An alternate solution is to deploy the application and MongoDB servers in a properly segmented network. When network segmentation is not possible, and a TLS approach is chosen, for instance, due to compliance requirements, one should use connection pools rather than opening and closing a connection for every operation to reduce the overhead.

References

[1] David Bermbach, Jorn Kuhlenkamp. Consistency in distributed storage systems. In International Conference on Networked Systems, Springer, Berlin, Heidelberg, 2013; 175-189.

[2] Eric A. Brewer. Towards robust distributed systems, In PODC 2000; 7(10.1145): 343477-343502.

[3] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story, Computer 2012; 45(2): 37-42.

[4] Werner Vogels. Eventually consistent, Communications of the ACM 2009; 52(1): 40-44.

[5] George Apostolopoulos, Vinod Peris, and Debanjan Saha. Transport Layer Security: How much does it really cost?, In IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies 1999; 2:717-725.

[6] Krishna Kant, Ravishankar Iyer, and Prasant Mohapatra . Architectural impact of secure socket layer on internet servers, In Proceedings International Conference on Computer Design 2000;7-14).

[7] Li Zhao, Ravi Iyer, Srihari Makineni, and Laxmi Bhuyan. Anatomy and performance of SSL processing, In IEEE International Symposium on Performance Analysis of Systems and Software; 2005:197-206.

[8] Cristian Coarfa, Peter Druschel, and Dan S. Wallach. Performance analysis of TLS Web servers, ACM Transactions on Computer Systems (TOCS) 2006; 24(1): 39-69.

[9] Satoshi Shirasuna, Aleksander Slominski, Liang Fang, and Dennis Gannon (2004, November). Performance comparison of security mechanisms for grid services. In Fifth IEEE/ACM international workshop on grid computing 2004; 360-364.

[10] Sergio Rapuano, and Eugenio Zimeo. Measurement of performance impact of ssl on ip data transmissions, Measurement 2008; 41(5): 481-490.

[11] Charles Shen, Erich Nahum, Henning Schulzrinne and Charles P. Wright. The impact of TLS on SIP server performance: Measurement and modelling, IEEE/ACM Transactions on Networking 2012: 20(4):1217-1230.

[12] Matjaz B. Juric, Ivan Rozman, Bostjan Brumen, Matjaz Colnaric, and Marjan Hericko. Comparison of performance of Web services, WS-Security, RMI, and RMI–SSL, Journal of Systems and Software 2016; 79(5): 689-700.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB, In Proceedings of the 1st ACM symposium on Cloud computing 2010; 143-154.

[14] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio Lopez and Billie Rinaldi, (2011, October). Ycsb++ benchmarking and performance debugging advanced features in scalable table stores, In Proceedings of the 2nd ACM Symposium on Cloud Computing 2011; 1-14.

[15] Liang Zhao, Anna Liu, and Jacky Keung. Evaluating cloud platform architecture with the care framework, In 2010 Asia Pacific Software Engineering Conference 2010; 60-69.

[16] David Bermbach, Liang Zhao, and Sherif Sakr. Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services, In Technology Conference on Performance Evaluation and Benchmarking 2013; 32-47.

[17] David Bermbach, and Stefan Tai. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behaviour, In Proceedings of the 6th Workshop on Middleware for Service Oriented Computing 2011; 1-6.

[18] Markus Klems, David Bermbach, and Rene Weinert. A runtime quality measurement framework for cloud database service systems, In Eighth International Conference on the Quality of Information and Communications Technology 2012; 38-46.

[19] Martin Grambow, Fabian Lehmann, and David Bermbach. Continuous benchmarking: Using system benchmarking in build pipelines, In IEEE International Conference on Cloud Engineering (IC2E) 2019; 241-246.

[20] Martin Grambow, Lukas Meusel, Erik Wittern, and David Bermbach (2020, March). Benchmarking microservice performance: a pattern-based approach, In Proceedings of the 35th Annual ACM Symposium on Applied Computing 2020; 232-241.

[21] Jitendra Kurmi and Suresh Prasad Kannojia. Comparative Study of SSL/TLS Cryptographic Libraries, International Journal of Innovative Research in Science, Engineering and Technology, 2021; 10(8): 11658-11662.

[22] Suresh Prasad Kannojia and Jitendra Kurmi. Analysis of Cryptographic Libraries (SSL/TLS), International Journal of Computer Sciences and Engineering 2021; 9(9); 59-62.