CASE STUDY

SYNOPSYS®

Data-Driven Software Security Insights

TLS Performance Overhead for MongoDB

TABLE OF CONTENTS

Abstract

The research presented within this document describes the precise performance overhead that TLS introduces when executing operations on one of the major NoSQL databases: MongoDB (about 45% in latency).

To explore TLS performance overhead for MongoDB, we recently conducted two tests simulating common database usage patterns. Our team first investigated connection pooling, where an application (or framework) uses a single connection for many database operations. Then, we considered one request per connection in which an application opens a connection, executes an operation, and immediately closes the connection after completing the operation.

Our findings conclude 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.

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. NoSQL databases fill this gap. These databases are being increasingly used to store sensitive information. Therefore, we can see that security is becoming a higher priority for firms using such databases.

Common attacks on database systems include eavesdropping (in which an attacker reads the communication with the database) and man-in-the-middle (MITM) attacks (in which an attacker spies on the communication and alters the information). 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 TLS/SSL. This prevents attackers from intercepting sensitive data transferred between the application and the NoSQL server. Unfortunately, enabling this feature reduces the communication performance with the NoSQL database server. Quantifying this performance degradation enables businesses to decide whether the security benefit is worth the additional cost.

It's possible to add TLS/SSL as a security feature. This prevents attackers from intercepting sensitive data.

In the following sections, we describe the precise performance overhead that TLS introduces when executing operations on MongoDB NoSQL database server. We conducted two tests simulating common database usage patterns, similar to those that Ernie Souhrada performed on MySQL:

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

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

Methodology and testing environment

When conducting the connection pool tests, we ran the Yahoo! Cloud System Benchmark (YCSB) 0.9.0 with Java 8 update 92 on a MongoDB instance with TLS enabled. We modified the MongoDB client used by the benchmark suite to properly use TLS.

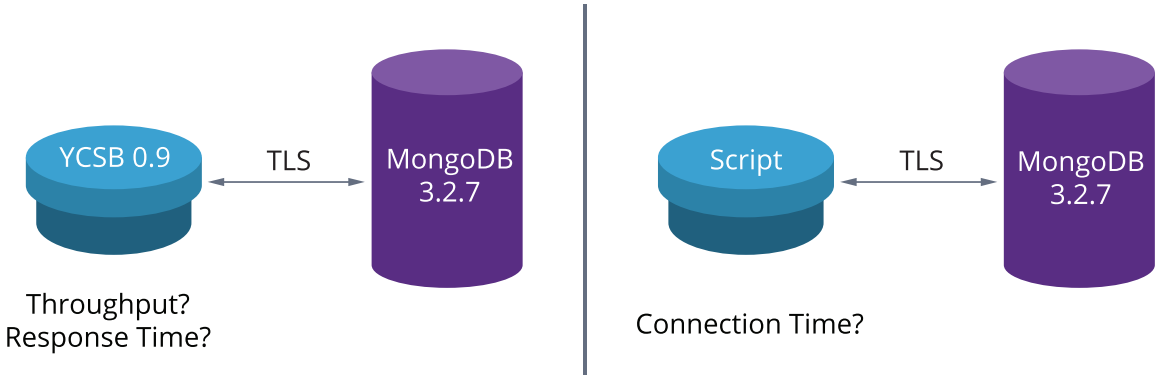
When conducting the one request per connection tests, we created a custom script using Node.js version 4.4.5.

We enabled TLS on MongoDB by following the steps outlined in the MongoDB documentation. Using TLS v1.2, 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). The server-side certificate was generated prior to the start of testing.

It could be argued that using an RSA handshake isn't an optimal approach to TLS, from both a security and performance perspective. Therefore, we looked into alternatives like the Diffie-Hellman key exchange or its elliptic curve variant. Unfortunately, MongoDB doesn't support these ciphers out of the box. For our tests to resemble a realistic scenario, we decided to maintain the original set-up.

It could be argued that using an RSA handshake isn't an optimal approach to TLS.

We performed all tests on an Intel Core i5-4300U 1.90GHz CPU (4 cores) with 12GB of RAM, running Windows 10 Pro. MongoDB version 3.2.7 and the testing tools were installed on this device. We also used a Samsung SSD 850 EVO 250GB storage drive.



Windows 10 Pro, 12 GB RAM, 4 cores, 240 GB SSD

Test #1: Connection pool

For this test, YCSB opens a connection to MongoDB, executes a test suite (many read operations) through the same connection, 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 operations.

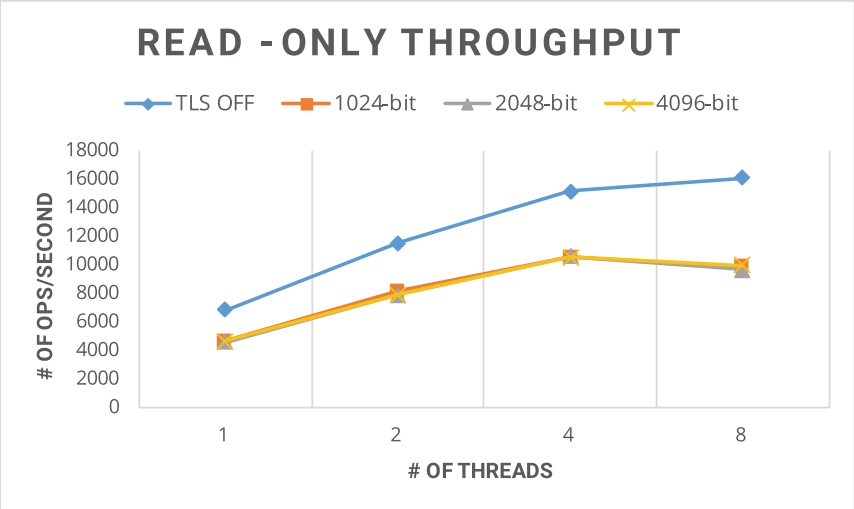
We compared throughput and response time between a connection without TLS and connections with TLS enabled (using 1024, 2048, or 4096-bit keys). To run the test, we used the following script:

```
1. #!/bin/bash
2. mongo ycsb --eval 'db.dropDatabase()' --sslCAFile public.crt --ssl --host localhost
3. bin/ycsb load mongodb -s -P workloads/workloadc -p recordcount=10000 -p mongodb.
   url=mongodb://localhost:27017 > tls_load_c.txt
4. for threads in 1 2 4 8 ;
5. do
6. bin/ycsb run mongodb -s -threads ${threads} -P workloads/workloadc -p
   operationcount=10000000 -p mongodb.url=mongodb://localhost:27017 > tls_run_c_${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 10,000 records using the C test suite (workloadc) from the YCSB benchmark (line 3). Subsequently, YCSB performs 10 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 four aforementioned cases, namely no TLS and TLS with 1024, 2048, and 4096-bit keys. It's important to note that we didn't use the URL parameter 'ssl=true' to enable TLS on a MongoDB connection since we made small changes to YCSB that enable TLS connections upon set-up.

When we examined the throughput, measured in number of operations per second, we noticed that enabling TLS introduces an overhead between 29% and 40%:

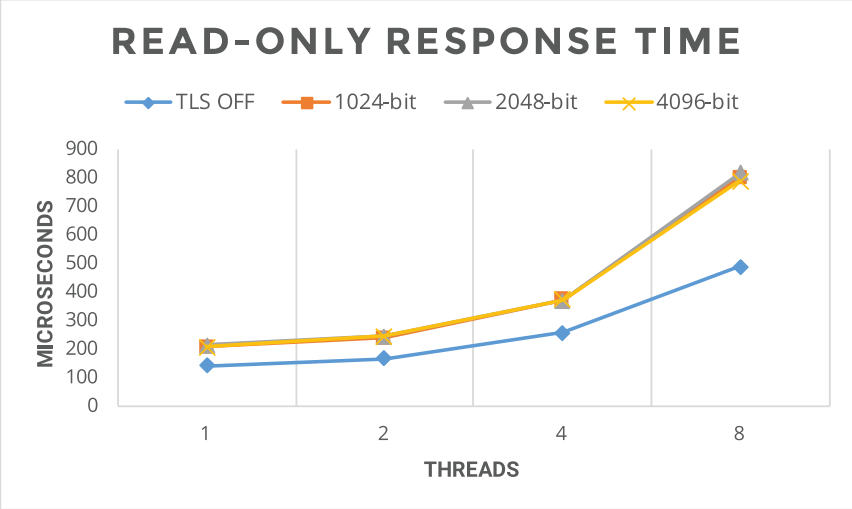


Within the raw data of this graph, the overhead percentage is presented in parentheses:

# of Threads	1	2	4	8
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 (29.90%)	9696.51 (39.96%)
4096-bit	4714.12 (31.40%)	7978.18 (31.21%)	10575.03 (30.43%)	10030.70 (37.89%)

Since the tests ran on a quad core machine, the slight setback of using eight threads was expected. 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 length 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 there are threads.

Let's examine the response time from the same test:



These results confirm previous findings. Within the raw data of this graph, the overhead percentage is presented in parentheses:

# of Threads	1	2	4	8
TLS OFF	143.59 (0.00%)	170.16 (0.00%)	260.35(0.00%)	490.17 (0.00%)
1024-bit	209.61 (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%)

Unfortunately, we weren't able to compute an accurate standard deviation. This was due to the fact that YCSB provides raw data in milliseconds. However, most operations and their averages (provided in microseconds) were under 1 millisecond.

Test #2: Connection time

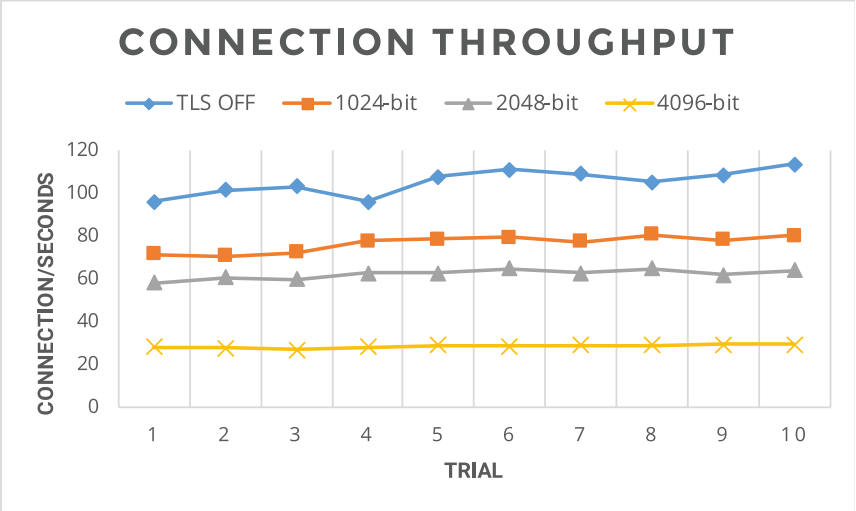
We expected that most performance overhead is introduced while making a new connection—despite the performance overhead of a connection pool (as measured in test #1). The second test examined this theory by measuring connection time using the following script:

We expected that most performance overhead is introduced while making a new connection.

```
var express = require('express');
var mongodb = require('mongodb');
var async = require('async');
var fs = require('fs');
var app = express();

var MongoClient = mongodb.MongoClient;
var url = 'mongodb://localhost:27017/?ssl=true';
var certFileBuf = fs.readFileSync('mongodb-cert.crt');
var options = {
      server: {sslCA: certFileBuf}
};

app.get('/run', function (req, res) {
      var times = [];
      async.timesSeries(10, function (n, callback) {
            ar start = new Date().getTime();
            async.timesSeries(100, function (n, callback) {
                  MongoClient.connect(url, options, function (err, db) {
                        db.close();
                        callback();
                  });
            }, function () {
                  var end = new Date().getTime();
                  times.push((end - start).toString());
                  callback();
            });
      }, function () {
            res.send(times);
      });

});

var server = app.listen(3000, function () {
      console.log('listening on port %d', server.address().port);
});
```

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

The following graph illustrates the elapsed time for each run and presents it as connections per second:



When extracting the raw data, we can see the average and standard deviation of the 10 consecutive runs. It's important to note an overhead moving from 27% for 1024-bit keys to 73% for 4096-bit keys with respect to a connection without TLS. Thus, 4096-bit TLS connections are up to four times slower to establish than unencrypted connections.

Encryption	Average connections per second	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

Analysis and conclusions

The results are similar to those of the MySQL experiment performed by Ernie Souhrada: TLS overhead is high, especially for applications using one connection per request with 4096-bit keys. For an application that maintains long-running connections, the difference between establishing a connection for different key lengths is negligible. There is, however, still a large overall overhead compared to unencrypted connections (around 45% in latency). Therefore, enabling TLS on a MongoDB database increases the throughput performance overhead between 29% to 40%. If such performance hit is not an option, TLS cannot be used 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.

THE SYNOPSYS DIFFERENCE

Synopsys offers the most comprehensive solution for integrating security and quality into your SDLC and supply chain. Whether you're well-versed in software security or just starting out, we provide the tools you need to ensure the integrity of the applications that power your business. Our holistic approach to software security combines best-in-breed products, industry-leading experts, and a broad portfolio of managed and professional services that work together to improve the accuracy of findings, speed up the delivery of results, and provide solutions for addressing unique application security challenges. We don't stop when the test is over. Our experts also provide remediation guidance, program design services, and training that empower you to build and maintain secure software.

For more information go to www.synopsys.com/software

SYNOPSYS®
185 Berry Street, Suite 6500
San Francisco, CA 94107 USA

U.S. Sales: (800) 873-8193
International Sales: +1 (415) 321-5237
Email: software-integrity-sales@synopsys.com