POLITECNICO DI TORINO

College of Computer Engineering, Cinema and Mechatronics

Master's Degree Thesis

Improvement of a system for retrieving and displaying systematic aircraft data

Supervisors
Prof. Bartolomeo Montrucchio
Dr. Antonio Costantino Marceddu

Candidate
Ahmad El Zein

December 2021

Summary

The aim or goal of this graduation thesis (Improvement of a system for retrieving and displaying systematic aircraft data) is the improvement of PhotoNext project.
The core of the project is the middleware created, that can be hosted in a Raspberry Pi 3 Model B board. The improvement is done by creating a middleware independent of the type of interrogator used. This independence will make the middleware support all types and sizes of interrogators that will be used on different sized airplanes.

The project is a collaboration between two departments of the Politecnico di Torino: DAUIN (Department of Control and Computer Engineering) and DIMEAS (Department of Mechanical and Aerospace Engineering).

Taking a general look on the project, three components can be seen:

1. **FBG Sensing System.**
   Where data is derived from, captured, analysed and collected.
2. **Cloud Network.**
   Where the collected data is stored.
3. **Viewer.**
   Where the collected data is simply displayed in real time and non-real-time mode.

The FBG Sensing System comprises the interrogator, the middleware and a set of FBG sensors that keep track of the unity (physical system).

The *Physical System* represents a unity being monitored by a set of FBG sensors, which they are a sort of distributed Bragg reflectors made up of short fiber segments that reflects specific wavelengths while transmitting all others.
The *Interrogator* is a device that can acquire information about the behavior of the optical fiber by sending laser beams to the Bragg grating and evaluating its response. The data collected may relate to different physical quantities depending on the interrogator used. They can provide information about wavelength, strain, temperature or pressure to which the fibers and thus the Bragg sensors are exposed. In this thesis a SmartScan© interrogator from SmartFibres© is used.
The *Middleware* is software that enables one or more kinds of communication or connectivity between two or more applications or application components in a distributed network. By making it easier to connect applications that weren't designed to connect with one another and providing functionality to connect them in intelligent ways. In this thesis the middleware is a C++ application that is used to retrieve data from sensors attached to the interrogator.
The *Cloud Network* is a computer network that connects cloud-based or cloud-enabled applications, services, and solutions over the internet. MongoDB is used to build the cloud network, it is an open source NoSQL database management program that doesn't require predefined schemas. It stores any type of data. This allows users to create any amount of fields in a document, allowing MongoDB databases scale more easily than relational databases.
The *Viewer* is a Desktop/AR Application developed in Unity, useful for displaying real-time and non-real-time data. The primary purpose of the viewer is to present data in a logical and understandable manner;however, the ultimate goal is to enable both real-time and offline data analysis using various methods. It reads data from a MongoDB instance or from the middleware directly

over TCP-IP, allowing it to work with a variety of configurations.

Previously, middleware was a dependent application with a monolithic architecture, which meant that it only worked with a single interrogator and that the functionally distinct aspects (e.g., data listening and parsing, data storage, error handling, and user interface) were all interwoven rather than being separated into architecturally distinct components (lack of Modularity). It is improved by making it independent of the type of interrogator used and removing the ability to send data directly to the viewer through TCP-IP connection. Other sub-improvements included overcoming various issues with the data preservation structures (buffer, queue, etc.) and figuring out how to reduce mongoDB server traffic, or how to interface with mongoDB once a specific amount of gathered items has been reached.

Acknowledgements

I'd like to extend my sincere gratitude to Professor Bartolomeo Montrucchio who trusted me and gave me the opportunity being his thesis candidate.

I'd like to sincerely thank Dr. Antonio Costantino Marceddu for his helpful and constructive recommendations, as well as his ongoing support and direction, patience, motivation, and extensive knowledge during the planning and implementation of this research project.

I'd like to express my grateful thanks to Eng. Matteo Dalla Vedova and Dr. Alessandro Aimasso for all of their support and assistance over the last few months.

I'd like to express my appreciation to the Department of Mechanical and Aerospace Engineering DIMEAS for allowing me utilize their equipment.

Finally, I'd like to express my heartfelt thanks to my family for their everlasting love, support, and assistance. My parents will always be responsible for providing me with the opportunities and experiences that have helped me become the person I am today.

Contents

List of Figures VI

1 Introduction 1

1.1 Overview . . . 1

1.2 MongoDB Technology . . . 2

1.2.1 MongoDB Definition . . . 2

1.2.2 MongoDB Architecture . . . 3

1.2.3 How It Works . . . 3

1.2.4 MongoDB Platforms . . . 3

1.2.5 MongoDB Pros and Cons . . . 4

1.3 Real-Time Technology . . . 4

1.3.1 Definition Of Term "Real-Time" . . . 4

1.3.2 Definition Of Real-Time Application . . . 4

1.4 Internet Communication Protocols . . . 4

1.4.1 Transmission Control Protocol/Internet Protocol (TCP/IP) . . . 4

1.4.2 User Datagram Protocol (UDP) . . . 6

1.5 Binary JavaScript Object Notation BSON Technology . . . 7

1.5.1 Definition Of BSON . . . 7

1.5.2 Understanding BSON . . . 7

1.5.3 BSON VS JSON . . . 7

1.5.4 Disadvantages . . . 8

2 System Architecture 9

2.1 System Overview . . . 9

2.2 FBG Sensing System . . . 9

2.2.1 Physical System Monitored By FBG Sensors . . . 9

2.2.2 Interrogator . . . 12

2.2.3 Middleware . . . 13

2.3 Cloud Network . . . 14

2.4 Viewer . . . 15

3 Proposed Solution 16
3.1 Overview . . . 16
3.2 Physical System . . . 17
3.3 Interrogator . . . 18
3.4 Middleware . . . 19
3.4.1 Middleware Classes Architecture . . . 20
3.4.2 Middleware Files Hierarchy . . . 22
3.4.3 Middleware CMake Files . . . 24
3.4.4 Middleware Client Class Modifications . . . 26
3.4.5 Middleware SmartScanInterrogator Class Modifications . . . 28
3.4.6 Middleware MongoDAO Class Modifications . . . 31
3.4.7 Middleware Data Models . . . 34
3.5 MongoDB Cloud . . . 37
3.5.1 What Is CRUD In MongoDB? . . . 38
3.5.2 How To Perform CRUD Operations. . . . 38
3.6 Viewer . . . 41
3.6.1 Real-Time Mode . . . 42
3.6.2 Non-Real-Time Mode . . . 43
3.6.3 Simulation's Outcome . . . 43

4 User Guide 45
4.1 MongoDB Compass . . . 45
4.1.1 What Is MongoDB Compass? . . . 45
4.1.2 What MongoDB Compass Can Do? . . . 45
4.1.3 Connect To MongoDB Using Compass . . . 48
4.1.4 Compass Home . . . 49

5 Test And Evaluation 51
5.1 Test Tools and Equipment . . . 51
5.1.1 Sensors . . . 51
5.1.2 SmartScan Interrogator . . . 52
5.1.3 Raspberry Pi 3 Model B . . . 53
5.1.4 Portable Computer . . . 54
5.2 Test Scenarios . . . 54
5.2.1 Connection Availability . . . 55
5.2.2 Middleware Memory Usage . . . 55
5.2.3 Data Rate And Middleware Stability . . . 57
5.2.4 Viewer Real Time Data Analysis . . . 59

6 Conclusion 62
6.1 Future Work . . . 62
6.1.1 Missing Features . . . 62
6.2 Conclusion . . . 63

Bibliography I

List of Figures

1.1 Architecture of the system [1]. . . . 2
1.2 Collection hierarchy [12]. . . . 2
1.3 MongoDB architecture [11]. . . . 3
1.4 Process of establishing a TCP connection (three-way handshake) [21]. . . . 5
1.5 UDP communication [24]. . . . 6
1.6 BSON format of storing data [32]. . . . 7
1.7 BSON encoding and decoding structure [32]. . . . 8

2.1 The Air Cargo Challenge aircraft launched by ICARUS [14]. . . . 10
2.2 Fiber Bragg grating sensor structure [15]. . . . 10
2.3 Fiber Bragg grating sensor [30]. . . . 11
2.4 FBG sensor embedded inside an ICARUS aircraft wing. . . . 11
2.5 SmartScan© from SmartFibres© [16]. . . . 12
2.6 SmartScan© with active FBG sensors connected [18]. . . . 12
2.7 SmartSoft Application Software. . . . 13
2.8 Raspberry Pi 3 Model B - 1GB RAM Board [19]. . . . 14
2.9 Cloud database and network [1]. . . . 14
2.10 Viewer graphical user interface. Import Model, HeatMap Color, Server/Network Configuration and Sensor Configuration menus are marked in red, green blue and orange respectively [3]. . . . 15

3.1 Monolithic architecture [26]. . . . 16
3.2 Physical system [1]. . . . 17
3.3 Aircraft by ICARUS team [27]. . . . 18
3.4 FBG interrogator [1]. . . . 18
3.5 SmartSoft Application Software. . . . 19
3.6 Middleware layer [1]. . . . 20
3.7 Old middleware architecture. . . . 21
3.8 New middleware architecture. . . . 22
3.9 Old middleware files hierarchy. . . . 23
3.10 New middleware files hierarchy. . . . 24
3.11 Piece of code of the old middleware CmakeLists.txt file. . . . 25

3.12 Piece of code of the new middleware CmakeLists.txt file. 25
3.13 Functions used by the old middleware Client class. 26
3.14 Functions used by the new middleware Client class. 27
3.15 A piece of code of the main function of the new middleware Client class. 27
3.16 Functions used by the old middleware SmartScanInterrogator class. 28
3.17 Functions used by the new middleware Listener class. 30
3.18 Functions used by the new middleware Parser class. 31
3.19 CleanPeakData structure. 31
3.20 Functions used by the Old Middleware MongoDAO Class. 32
3.21 MongoDAO's InsertUnityData() function. 32
3.22 Functions used by the New Middleware MongoDAO Class. 33
3.23 MongoDAO's insertMultipleData() function. 33
3.24 RawData structure. 34
3.25 PeakData structure. 35
3.26 CleanPeakData structure. 35
3.27 Configuration data structure. 36
3.28 CleanPeakData data document model. 37
3.29 Configuration data document model. 37
3.30 Return object of insertOne() operation. 39
3.31 Return object of insertMany() operation. 39
3.32 Return object of updateOne() operation. 40
3.33 Return object of updateMany() operation. 40
3.34 Return object of replaceOne() operation. 40
3.35 Return object of deleteOne() operation. 41
3.36 Return object of deleteMany() operation. 41
3.37 Home view of the Desktop application 42
3.38 Real-Time Server/Network Configuration menu 43
3.39 Non-Real-Time Server/Network Configuration menu. 43
3.40 Example of a Log file with three active sensors [3] 44
3.41 Example of a line graph image with 64 active sensors [3] 44

4.1 Importing data from CSV file [35]. 46
4.2 Querying on data [35]. 46
4.3 Creating aggregation pipelines [35]. 47
4.4 Running commands in a shell [35]. 47
4.5 Pasting the connection string [35]. 48
4.6 Filling connection fields [35]. 48
4.7 Hostname dialog fields. 49
4.8 Compass home screen [35]. 50

5.1 Thermometric probe. 51
5.2 Tension bar. 52
5.3 ICARUS aircraft wing. 52
5.4 4 active channels SmartScan© interrogator. 53
5.5 Raspberry Pi 3 Model B. 53
5.6 Portable computer launching the viewer and MongoDB Compass. 54
5.7 Timestamps stored during the simulation [3] 54
5.8 Connection established between the interrogator and the middleware. 55
5.9 Raspberry Pi 3 Model B memory usage without launching the middleware. 56
5.10 Raspberry Pi 3 Model B memory usage after launching the old middleware. 56
5.11 Raspberry Pi 3 Model B memory usage after launching the new middleware. 56
5.12 Total data sent in a 4 hours and a half period of time. 57
5.13 Total data stored in a 4 hours and a half period of time. 58
5.14 Full system under test. 59
5.15 Raspberry Pi 3 Model B. 60
5.16 Model with 8 active gratings under real-time mode test. 61

6.1 Microsoft© HoloLens AR system [36]. 63
6.2 AR/VR visualisation and analysis [1]. 63

Chapter 1

Introduction

The following chapter provides an overview of the proposed framework as well as a brief description of the system architecture and the relevant technologies used.

1.1 Overview

The aim or goal of this graduation thesis (Improvement of a system for retrieving and displaying systematic aircraft data) is the improvement of PhotoNext project. The project was created to study and monitor an aircraft constructed by the Politecnico di Torino's Innovation Center for Amateur Rocketry and Unmanned Ships (ICARUS) team as part of their "ANUBIS" project.

The project is a collaboration between two departments at Politecnico di Torino, DIMEAS (Department of Mechanical and Aerospace Engineering) and DAUIN (Department of Control and Computer Engineering), as part of the POLITO Inter-Departmental Center for Photonic technologies (PhotoNext).

The core of the project is the middleware created, that can be hosted in a Raspberry Pi 3 Model B board. This improvement is done by creating a middleware independent of the type of interrogator used. This independence will make the middleware support all types and sizes of interrogators that will be used on different sized airplanes.

Taking a general look on the project, three components can be seen:

1. **FBG Sensing System**
   Where data is derived from, captured, analysed and collected.

2. **Cloud Network**
   Where the collected data is stored.

3. **Viewer**
   Where the collected data is simply displayed in real time and non-real-time mode.

The FBG Sensing System comprises the interrogator, the middleware and a set of FBG sensors that keep track of the unity (physical system).

A Cloud Network is a computer network that connects cloud-based or cloud-enabled applications, services, and solutions over the internet.

A Viewer is a Desktop/AR Application developed in Unity, useful for displaying real-time and non-real-time data.

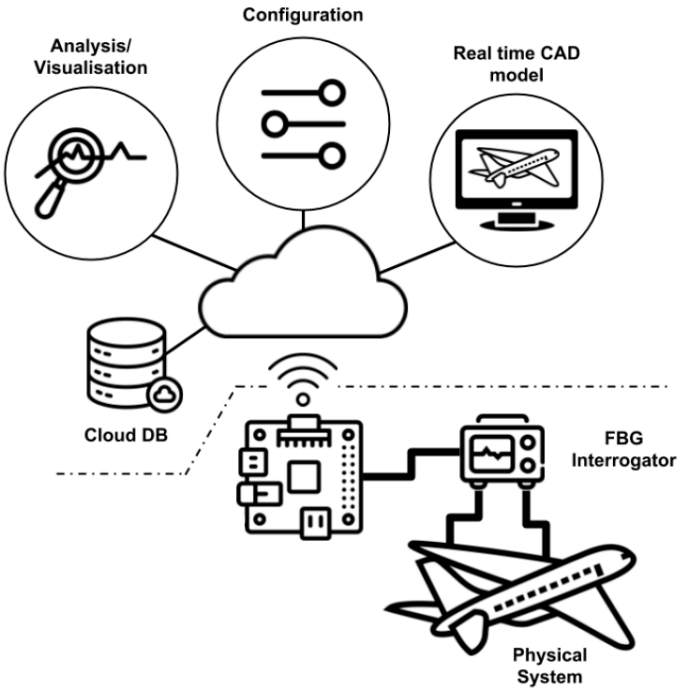

Figure 1.1. Architecture of the system [1].

1.2 MongoDB Technology

1.2.1 MongoDB Definition

MongoDB is an open source NoSQL database management program. Traditional relational databases are replaced by NoSQL databases. NoSQL databases are quite useful for working with large sets of distributed data. MongoDB is a tool that can manage document-oriented information, store or retrieve information.
MongoDB can handle a wide range of data types. It's one of numerous non-relational database technologies that arose in the mid-2000s under the NoSQL banner, usually for use in big data applications and other processing operations involving data that doesn't fit well in a conventional relational paradigm. MongoDB's design is made up of collections and documents rather than tables and rows, like in relational databases.[10]

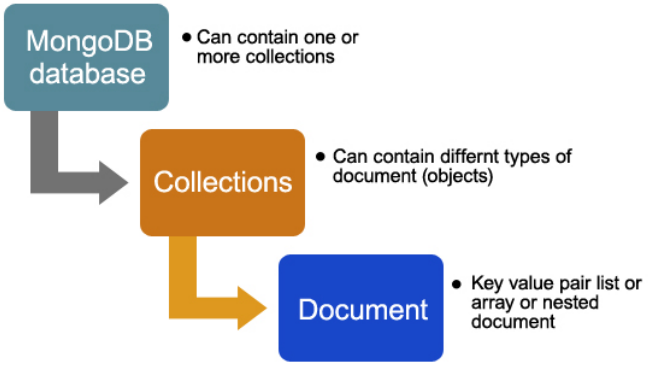

Figure 1.2. Collection hierarchy [12].

1.2.2 MongoDB Architecture

There are three main parts to the MongoDB technology:

1. **MongoDB Server**
   It is commonly known as mongod daemon, the primary process which handles data requests, manages data format, and performs background management operations. There can be many mongod daemons running as primary secondary instances.
2. **MongoDB Mongos**
   It is the routing controller service that routes information and data in the cluster.
3. **MongoDB Shell**
   It is the interactive interface. By using JavaScript to command, the developer can examine results of queries and check test cases.

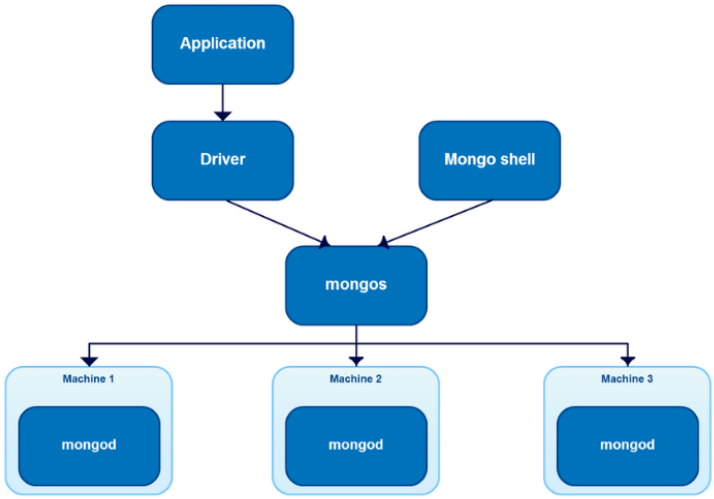

Figure 1.3. MongoDB architecture [11].

1.2.3 How It Works

MongoDB works using records, which are documents that contain a data structure made up of field and value pairs. MongoDB's basic data unit is the document. The documents resemble JavaScript Object Notation, although they employ a binary JSON variation (BSON). The advantage of utilizing BSON is that it can handle a wider range of data formats. These documents have fields that are similar to columns in a relational database. According to the MongoDB user documentation, the values contained can be a number of data formats, including other documents, arrays, and arrays of documents. A main key will be used as a unique identifier in documents. Collections are groups of documents that work in the same way that relational database tables do. Collections can hold any form of data; however, the data in a collection cannot be dispersed across many databases.

1.2.4 MongoDB Platforms

MongoDB is available in community and commercial versions through vendor MongoDB Inc. MongoDB Community Edition is the open source release, while MongoDB Enterprise Server brings added security features, an in-memory storage engine, administration and authentication features, and monitoring capabilities through Ops Manager.

A Graphical User Interface (GUI) called MongoDB Compass gives users a way to work with document structure, conduct queries, index data and more. The MongoDB Connector for BI allows users to connect the NoSQL database to their business intelligence tools to visualize data and create reports using SQL queries.

1.2.5 MongoDB Pros and Cons

Like other NoSQL databases, MongoDB doesn't require predefined schemas. It stores any type of data. This allows users to create any amount of fields in a document, allowing MongoDB databases scale more easily than relational databases.
One of the advantages of using documents is that these objects map to native data types in a number of programming languages. Also, having embedded documents reduces the need for database joins, which can reduce costs.
Though there are some valuable benefits to MongoDB, there are some downsides to it as well. In a MongoDB cluster, a user only needs to put up one master node thanks to its automatic failover method. If the master fails, another node will become the new master automatically. This changeover ensures continuity, but it isn't quick, taking up to a minute to complete.[10]

1.3 Real-Time Technology

1.3.1 Definition Of Term "Real-Time"

It's the same thing as stating something is happening "live" or "on-the-fly" when it happens in real time. This means that the information is updated so quickly that the user does not notice any delay.

1.3.2 Definition Of Real-Time Application

A Real-Time Application (RTA) is a program that operates in a time period that the user perceives to be immediate or current. The latency must be under a certain threshold, commonly measured in seconds. The Worst-Case Execution Time (WCET), or the highest amount of time a specific task or collection of tasks required on a given hardware platform, determines whether or not an application qualifies as an RTA.The use of RTAs is called Real-Time Computing (RTC).[13]

1.4 Internet Communication Protocols

A communication protocol is a set of rules that allows two or more entities in a communications system to send data using any physical quantity variation. The protocol specifies the communication rules, syntax, semantics, and synchronization, as well as error recovery techniques. Hardware, software, or a combination of both can implement protocols.[20]

1.4.1 Transmission Control Protocol/Internet Protocol (TCP/IP)

1.4.1.1 What is TCP/IP?

Transmission Control Protocol (TCP) is connection-oriented, meaning that data can be transmitted in both directions once a link has been established. TCP contains built-in systems to check for faults and ensure that data is delivered in the order it was transmitted, making it ideal for transferring data such as still photos, data files, and web pages.
But while TCP is instinctively reliable, its feedback mechanisms also result in a larger overhead, translating to greater use of the available bandwidth on the network.

1.4.1.2 How a TCP/IP connection is established?

The following are the prerequisites for establishing a successful TCP connection: Both endpoints must have a unique IP address (IPv4 or IPv6) and the necessary data transfer port assigned and configured. The IP address is used as an identifier, whereas the port is used by the operating system to assign connections to specific client and server programs.
The actual process for establishing a connection with the TCP protocol is as follows:

1. The requesting client sends the server a SYN packet or segment (SYN stands for synchronise) with a unique, random number. This number ensures full transmission in the correct order (without duplicates).

2. If the server has received the segment, it agrees to the connection by returning a SYN-ACK packet (ACK stands for acknowledgment) including the client's sequence number plus 1. It also transmits its own sequence number to the client.

3. The client acknowledges the receipt of the SYN-ACK segment by sending its own ACK packet, which in this case contains the server's sequence number plus 1. At the same time, the client can already begin transferring data to the server.

Since the TCP connection is established in three steps, the connection process is called a three-way handshake [21].

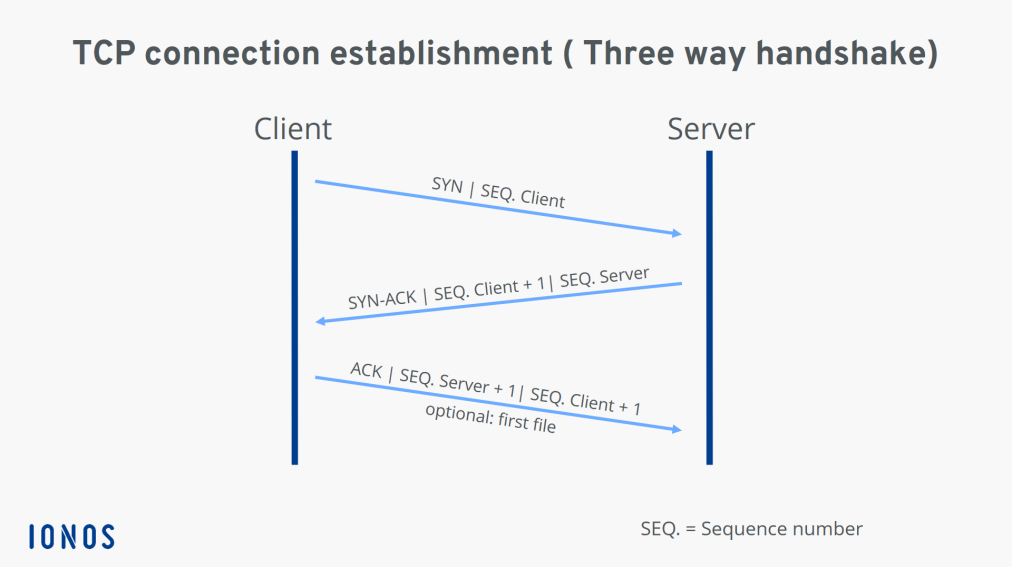

Figure 1.4. Process of establishing a TCP connection (three-way handshake) [21].

1.4.1.3 Features of TCP/IP [22]

Here are some important features of TCP/IP:

- Delivery acknowledgements.
- Delays transmission when the network is congested.
- Easy error detection.

1.4.2 User Datagram Protocol (UDP)

1.4.2.1 What is UDP?

The User Datagram Protocol (UDP) is a communications protocol that is mainly used on the internet to establish low-latency and loss-tolerant connections between applications. UDP speeds up transfers by allowing data to be transferred before the receiving party has agreed. As a result, UDP is advantageous in time-critical communications such as voice over IP (VoIP), DNS search, and video or audio playback. The User Datagram Protocol (UDP) is an alternative to the Transmission Control Protocol (TCP) [23].

1.4.2.2 How UDP works?

UDP uses IP to get a datagram from one computer to another. UDP works by gathering data in a UDP packet and adding its own header information to the packet. This data consists of the source and destination ports on which to communicate, the packet length and a checksum. After UDP packets are encapsulated in an IP packet, they're sent off to their destinations.
Unlike TCP, UDP doesn't guarantee the packets will get to the right destinations. This means UDP doesn't connect to the receiving computer directly, which TCP does. Rather, it sends the data out and relies on the devices in between the sending and receiving computers to correctly get the data where it's supposed to go.

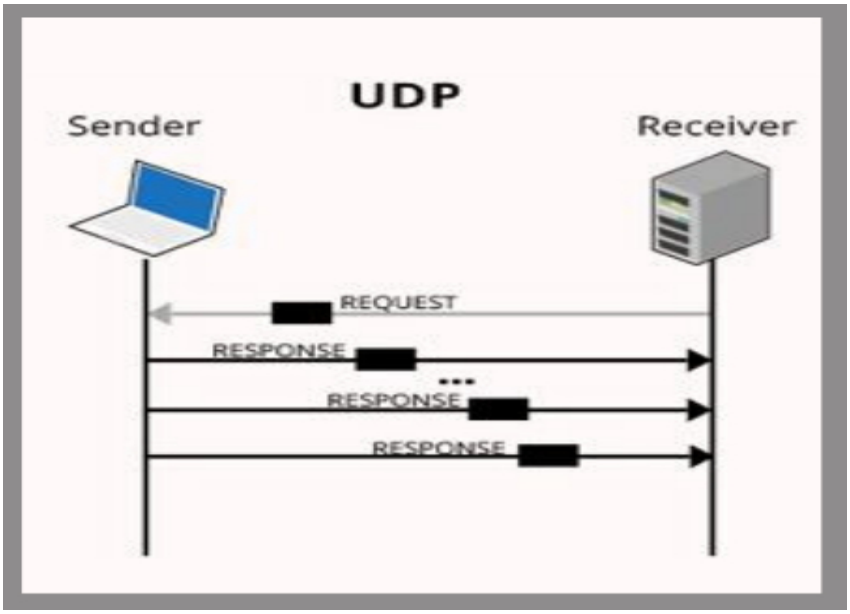

Figure 1.5. UDP communication [24].

1.4.2.3 Features of UDP [22]

Here are some important features of UDP:

- Supports bandwidth-intensive applications that tolerate packet loss.
- Less delay.
- It sends the bulk quantity of packets.
- Possibility of the data loss.
- Allows small transaction (DNS lookup).

1.5 Binary JavaScript Object Notation BSON Technology

1.5.1 Definition Of BSON

BSON simply stands for "Binary JSON," and that's exactly what it was invented to be. BSON's binary structure encodes type and length information, which allows it to be parsed much more quickly. Since its initial formulation, BSON has been extended to add some optional non-JSON-native data types, like dates and binary data, without which MongoDB would have been missing some valuable support. Languages that support any kind of complex mathematics typically have different sized integers (ints vs longs) or various levels of decimal precision (float, double, decimal128, etc.). Not only is it helpful to be able to represent those distinctions in data stored in MongoDB, it also allows for comparisons and calculations to happen directly on data in ways that simplify consuming application code [31].

1.5.2 Understanding BSON

BSON does not in any way replace JSON but in fact an Interchange Format is particularly used in storage and as a network transfer format used in the MongoDB database. BSON can be thought of as a super set of JSON, consisting of all properties that JSON has with some additional features:

- **LightWeight**
  BSON is designed to have minimum space overhead which is preferred for any data representation format over a network.
- **Easily Traversable**
  This is one of its most important feature and the reason why MongoDB uses BSON,is that query can be accessed in lesser time than before.
- **Less Time Required For Encoding Or Decoding**
  The data types used by BSON are easy to encode and decode which ensures faster operation capabilities [32].

1.5.3 BSON VS JSON

Binary JSON differs from JSON depending on many different categories like:

1. **Type Of File**
   JSON is used to store files in a standard file format that include both human and machine readable text, whereas BSON only stores binary files that can be read by machines.
2. **Structure**
   BSON is made up of a list of ordered elements with fieldnames, field types, and field values.

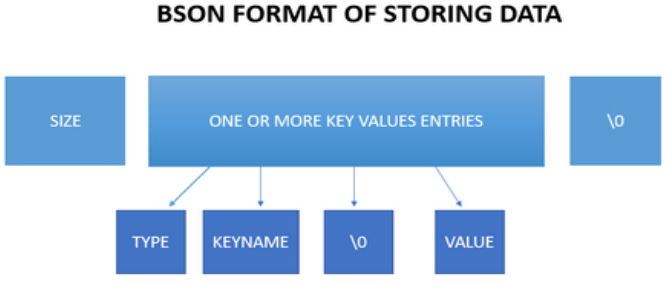

Figure 1.6. BSON format of storing data [32].

3. **Encoding And Decoding Techniques**
   Since BSON is a serialization format, decoding BSON and re-encoding JSON is required. BSON is constructed in such a way that it has a faster encoding and decoding technique and uses more space than JSON for smaller integers, but it is still much faster to parse than JSON.

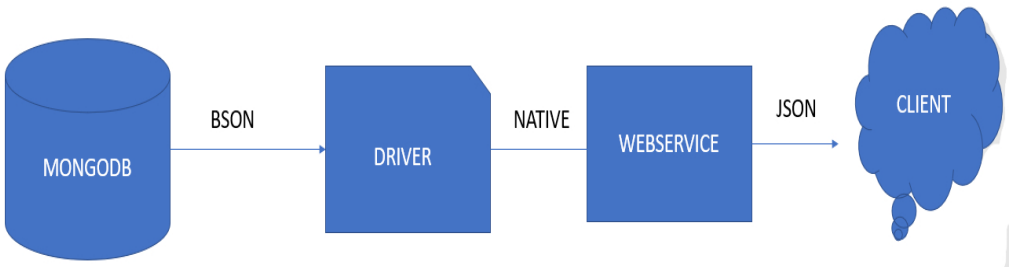

Figure 1.7. BSON encoding and decoding structure [32].

4. **Traversal**
   Unlike JSON, BSON does not scan through the entire file, but instead indexes the important content and skips the rest.

5. **Data Types**
   The availability of extra data types such as Date and BinData data types, in addition to the basic data types like numbers, characters, and other Boolean values, is the main advantage of BSON over JSON.
   BSON offers a variety of data types, including:

   (a) double (64-bit IEEE 754 floating point number)
   (b) date (integer number of milliseconds since the Unix epoch)
   (c) byte array (binary data)
   (d) BSON object and BSON array
   (e) MD5 binary data

1.5.4 Disadvantages

BSON has a slight disadvantage when it comes to memory efficiency as it requires some overhead to create additional fields like field name, field value etc [32].

Chapter 2

System Architecture

This chapter delves into the system's overall architecture.

2.1 System Overview

The purpose of this thesis is the enhancement of PhotoNext project. This enhancement is done by creating a middleware independent of the type of interrogator used. This independence will make the middleware support all types and sizes of interrogators that will be used on different sized airplanes. Other enhancements were made to the middleware's architecture, data filtering and collection, and data transfer to the cloud server methods, which will be discussed later.

Examining the project's system in deeper level, particularly the FBG Sensing System, three sub-components will be added to the existing system architecture.

1. **FBG Sensing System.**
   - Physical system monitored by FBGs sensors
     Where data is derived from.
   - Interrogator
     Where data is captured.
   - Middleware
     Where data is analysed and collected.
2. **Cloud Network.**
   Where the collected data is stored.
3. **Viewer.**
   Where the collected data is simply displayed in real time and non-real-time mode.

The next sections will go over each component and its sub-components.

2.2 FBG Sensing System

2.2.1 Physical System Monitored By FBG Sensors

A network of fibre Bragg grating sensors or transducers embedded within or attached to the structure being monitored.

2.2.1.1 Physical System

A physical system represents a unity being monitored by a set of FBG sensors. It may be any system that requires continuous temperature (or displacement) measurements at specific points in its structures.

Figure 2.1. The Air Cargo Challenge aircraft launched by ICARUS [14].

2.2.1.2 Fiber Bragg Grating Sensors

A fiber Bragg grating (FBG) is a sort of distributed Bragg reflector made up of short fiber segments that reflects specific wavelengths while transmitting all others. This is accomplished by varying the refractive index of the fibre core on a periodic basis, resulting in a wavelength specific dielectric mirror. To block certain wavelengths, a fiber Bragg grating can be utilized as an inline optical filter.

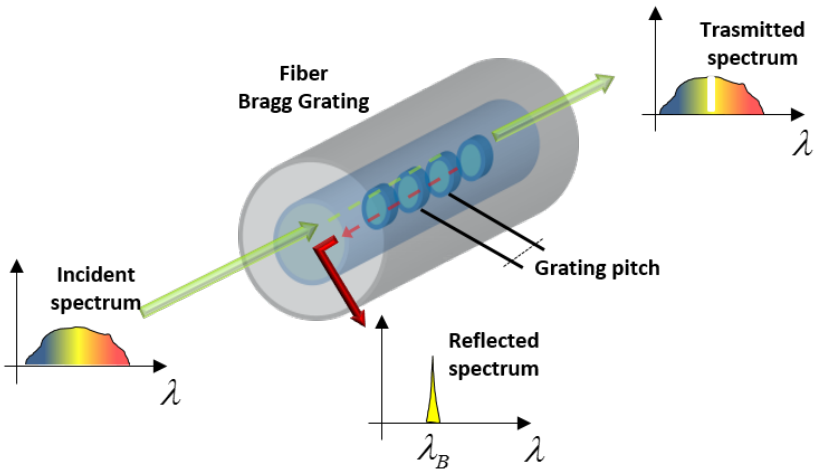

Figure 2.2. Fiber Bragg grating sensor structure [15].

FBG sensors are a proven structural health monitoring technology utilized for the in situ monitoring of advanced structures in aviation, aerospace systems, civil structures, and the petrochemical industry. It can be easily cast, implanted, or surface mounted on a structure due to its lightweight, micron-size transducers, and immunity to electromagnetic interference. To fulfill extreme environmental conditions and structural criteria for a variety of applications, FBG sensors can be manufactured in glass or plastic optical fiber. To resist difficult settings such as integrated in fiber composite structures, transatlantic fiber cable installations, tow-array sonar, and missile fiber guiding systems, the sensor fibers are wrapped in robust, durable materials.

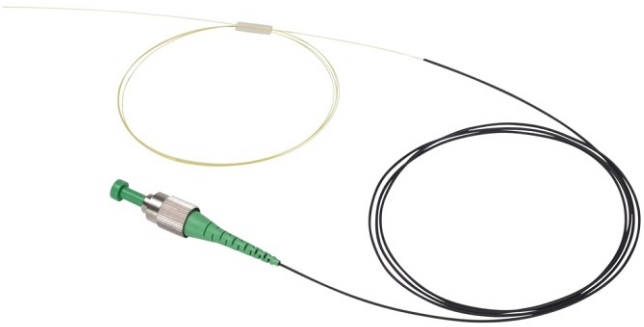

Figure 2.3. Fiber Bragg grating sensor [30].

The fibre Bragg grating is a significant optical component that is gaining traction in a variety of disciplines of optical technology, including optical fiber communication and sensing.[8] They have gained rapid acceptance in aerospace and automotive structural health monitoring applications for the measurement of strain, stress, vibration, acoustics, acceleration, pressure, temperature,moisture, and corrosion distributed at multiple locations within the structure using a single fiber element.

Figure 2.4. FBG sensor embedded inside an ICARUS aircraft wing.

The most prominent advantages of FBGs are: small size and light weight, multiple FBG transducers on a single fiber, and immunity to radio frequency interference.
A major disadvantage of FBG technology is that conventional state-of-the-art fiber Bragg grating interrogation systems are typically bulky and heavy bench top instruments that are assembled from off-the-shelf fiber optic and optical components integrated with a signal electronics board

into an instrument console.

2.2.2 Interrogator

An interrogator is a device that can acquire information about the behavior of the optical fiber by sending laser beams to the Bragg grating and evaluating its response. The data collected may relate to different physical quantities depending on the interrogator used. They can provide information about wavelength, strain, temperature or pressure to which the fibers and thus the Bragg sensors are exposed. A SmartScan© interrogator from SmartFibres© is used in this work.

2.2.2.1 SmartScan© Description

SmartScan© is a dynamic interrogator for use with fiber Bragg grating (FBG) sensors. SmartScan© interrogator provides both raw and peak data and can read 64 different FBGs (4 channels, 16 gratings per channel). It also has an RJ45 Ethernet port, a serial connector for diagnostics/servicing, and 2 USB ports.

Figure 2.5. SmartScan© from SmartFibres© [16].

2.2.2.2 SmartScan© Operation

SmartScan© has an electrically tuneable laser that generates light on multiple fibers at 400 distinct wavelengths with a 40 nm bandwidth. The light reflected from each fibre at each of the 400 laser wavelengths is then measured by optical detectors inside SmartScan©, resulting in a spectrum of the connected FBGs.

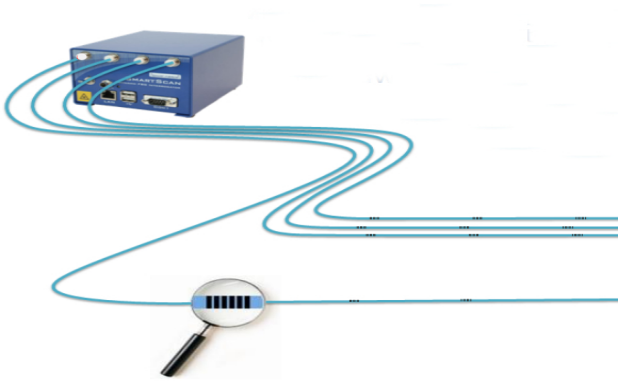

Figure 2.6. SmartScan© with active FBG sensors connected [18].

2.2.2.3 SmartScan© Usage Benefits

SmartScan© most essential use advantages are:

1. **Robustness**
   - The light source in SmartScan© is a widely tuneable semiconductor laser with active optical elements packed in a small monolithic chip that is only a few centimetres long. The rest of the components are optical and electrical passives. SmartScan© is therefore much more resistant to heat and mechanical effects than sweeping laser or spectrometer-based sensors. Several military flight demonstrations have shown the instrumentation's reliability.
2. **Speed**
   - Data rates of 25 kHz are possible for sequentially sampled FBGs with SmartScan©'s agile laser and specialized tuning circuits. Optical sensors can now be used to analyze high-speed vibration and other similar phenomena.
3. **Resolution**
   - At 2.5 kHz, SmartScan© achieves sub-picometer resolution per sample. This enables remarkable resolution through oversampling and averaging [18].

2.2.2.4 SmartFibres© Software

SmartFibres also offers SmartSoft Application Software, a Microsoft Windows application that interacts with the interrogator over UDP.

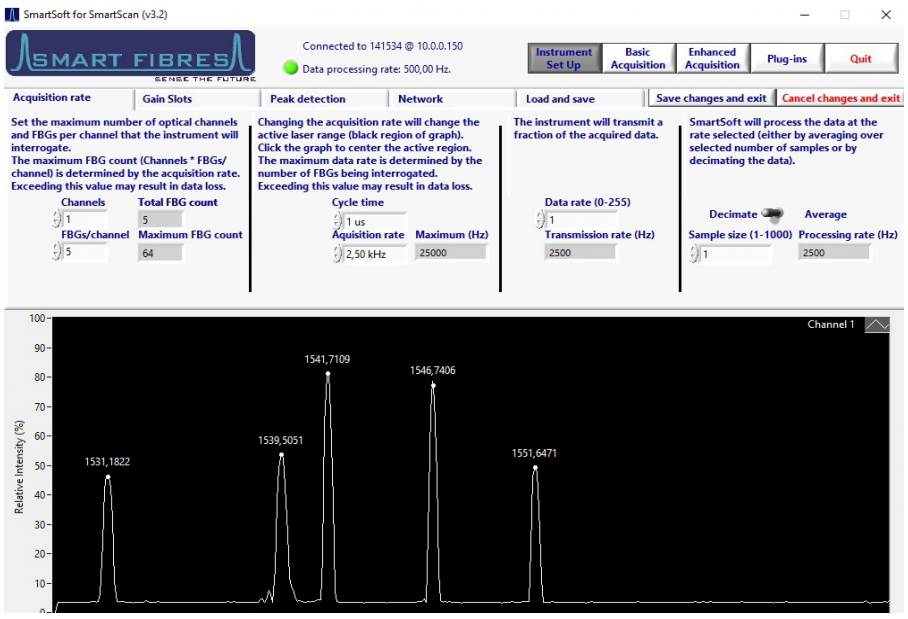

Figure 2.7. SmartSoft Application Software.

2.2.3 Middleware

A middleware is software that enables one or more kinds of communication or connectivity between two or more applications or application components in a distributed network. By making it easier to connect applications that weren't designed to connect with one another and providing

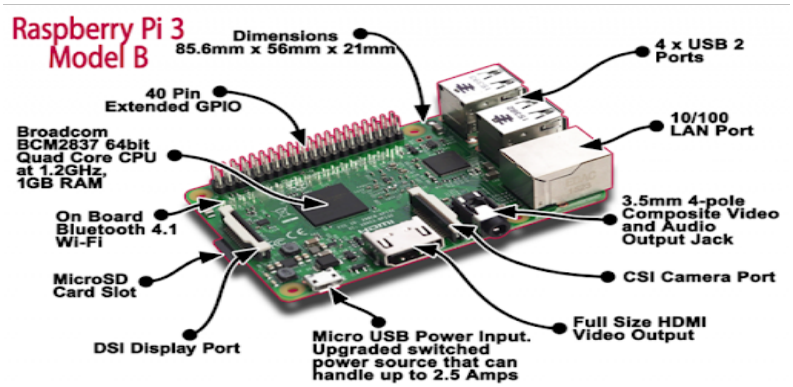

Figure 2.8. Raspberry Pi 3 Model B - 1GB RAM Board [19].

functionality to connect them in intelligent ways. The middleware, which is a C++ application running on a Raspberry Pi 3 Model B, is the core of the project.
The Raspberry Pi board will be connected to the interrogator over Ethernet after the middleware has been deployed. The middleware will serve as a standby listener for any sensor data received from the interrogator, filtering, collecting, and analyzing the data. A remote database server is linked with the middleware. As a result, the middleware will only send useful and purposeful data to the distant database server after gathering and processing them.
Considering useful and purposeful data implying that just the peak data is taken into account, since raw data gathering and analysing is discarded in this improvement.

2.3 Cloud Network

MongoDB was used to build the cloud network, it is an open source NoSQL database management program that doesn't require predefined schemas. It stores any type of data. This allows users to create any amount of fields in a document, allowing MongoDB databases scale more easily than relational databases. MongoDB's design is based on collections and documents rather than tables and rows, as is the case with relational databases, which lowers the need for database joins, thus reduces costs.

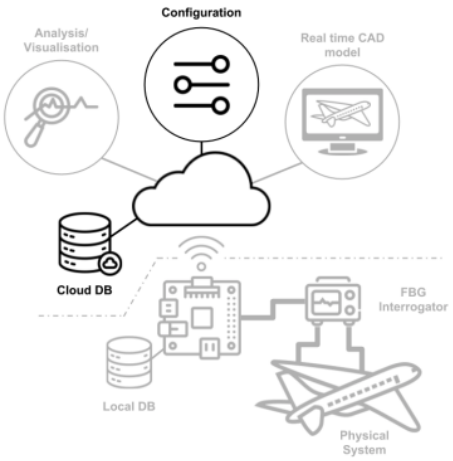

Figure 2.9. Cloud database and network [1].

2.4 Viewer

The primary purpose of the Viewer is to present data in a logical and understandable manner; however, the ultimate goal is to enable both real-time and offline data analysis using various methods. The Viewer is a Desktop/AR Application developed in Unity, that can read data from a MongoDB instance or from the middleware directly over TCP-IP, allowing it to work with a variety of configurations.

The upgraded middleware removes the ability to send data directly to the viewer through TCP-IP connection. As a result, the viewer will interact directly with the MongoDB server in one of two modes:

1. **Real-Time**
   Using Change Stream on a MongoDB instance.
2. **Non Real-Time**
   Reading data from a MongoDB instance that has already been recorded.

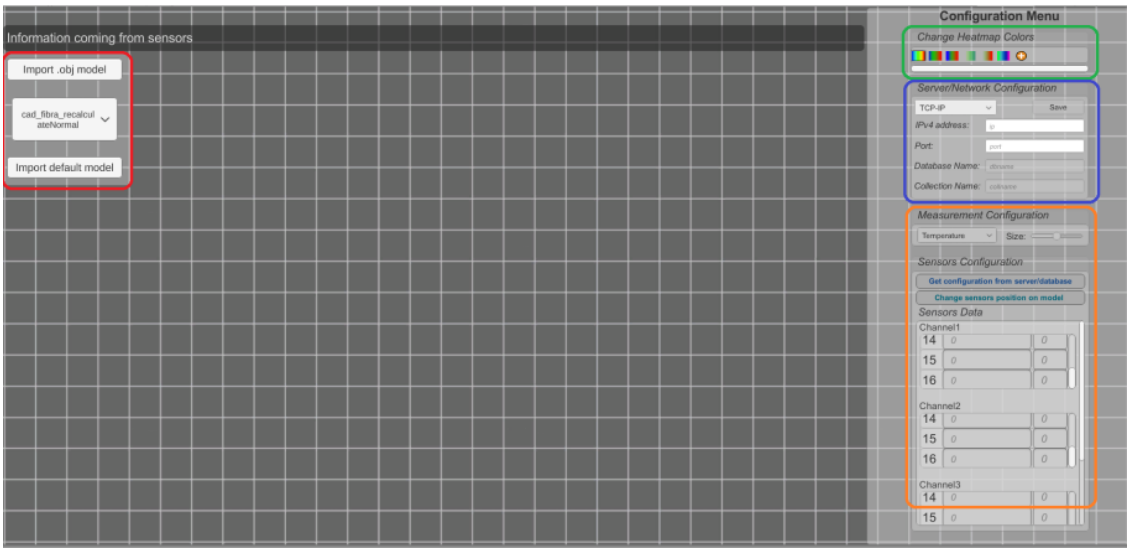

Figure 2.10. Viewer graphical user interface. Import Model, HeatMap Color, Server/Network Configuration and Sensor Configuration menus are marked in red, green blue and orange respectively [3].

Chapter 3

Proposed Solution

The main goal of this thesis was to improve the middleware, thus this chapter go over the modifications introduced to its architecture, classes, methods, and variables.

3.1 Overview

Earliar, the middleware used to be a dependent application with a monolithic architecture, so that, it works only with a particular interrogator and the functionally distinct aspects (e.g., data listening and parsing, data storage, error handling, and user interface) were all interwoven rather than containing architecturally distinct components (lack of Modularity).

What is a Monolithic Application?
A monolithic application is a single-tiered software application that combines the user interface and data access code into a single program that runs on a single platform.
A monolithic application is self-contained and independent from other computing applications. The design idea is that the application is responsible for not only doing a certain goal, but also for performing all of the steps required to fulfill that task [25].

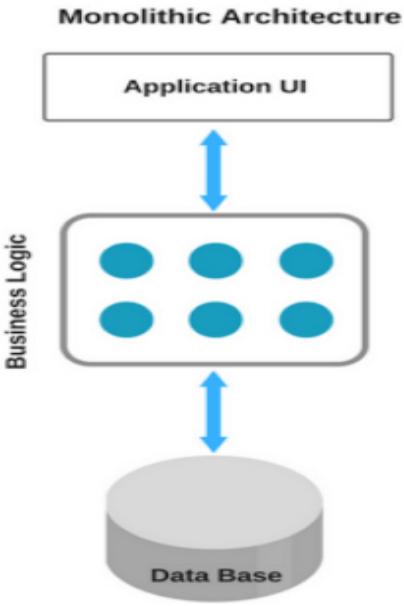

Figure 3.1. Monolithic architecture [26].

What is Modularity?
Modularity is desirable, in general, as it supports reuse of parts of the application logic and also facilitates maintenance by allowing repair or replacement of parts of the application without requiring wholesale replacement.

The middleware's architecture is improved, and it is divided into different classes, each with its own set of tasks to do in terms of listening, parsing, and storing data.
As upgrading the middleware, certain features are deleted and new ones are introduced.

- Added features:
  - Interrogator independency.
  - Storing only relevant configuration and peak data.
  - Multiple data insertion into MongoDB database.
  - A count of active gratings and the overall amount of data sent to MongoDB database.
- Removed features:
  - Raw data listening, parsing and storing.
  - The Viewer's ability to read data from the middleware directly through TCP-IP.
  - Single data insertion into MongoDB database.

3.2 Physical System

The physical system respresents the unity that FBG sensors are monitoring. It may be any system that requires continuous temperature (or displacement) measurements at specific points in its structures. These measurements may be required as part of a preventative plan to identify structural concerns in a system or to get precise data on physical events occurring within the monitored object.

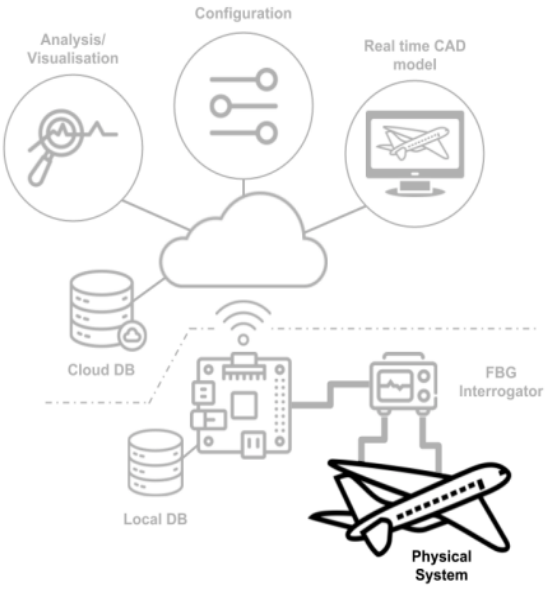

Figure 3.2. Physical system [1].

Depending on the application, real-time data (within a few seconds of collection) or stored data for additional off-line analysis may be required. For this thesis , the DIMEAS department will

integrate FBG sensors into an Unmanned Aerial Vehicle (UAV) created by the ICARUS team of the Politecnico di Torino. In order to detect abnormalities and deviations, data from the sensors must be retrieved at a high-speed rate. This test scenario necessitates the use of a specific interrogator that is robust enough for aerospace operations.

Figure 3.3. Aircraft by ICARUS team [27].

3.3 Interrogator

As previously stated, this test scenario involves the employment of a particular interrogator capable of handling aircraft operations. SmartScan© interrogator is used because it has an electrically tuneable laser that generates light on multiple fibers at 400 distinct wavelengths with a 40 nm bandwidth and because sequentially sampled FBGs can achieve data rates of 25 kHz with SmartScan©'s agile laser. Optical sensors can now be used to analyse high-speed vibration and other similar phenomena.

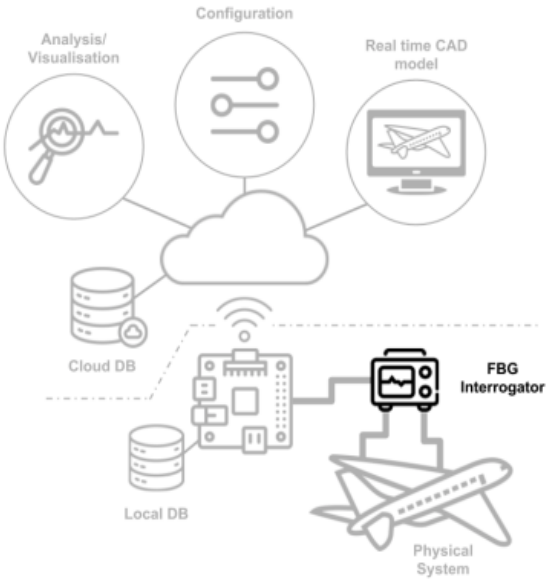

Figure 3.4. FBG interrogator [1].

SmartFibres also offers SmartSoft Application Software, a Microsoft Windows application that interacts with the interrogator over UDP.

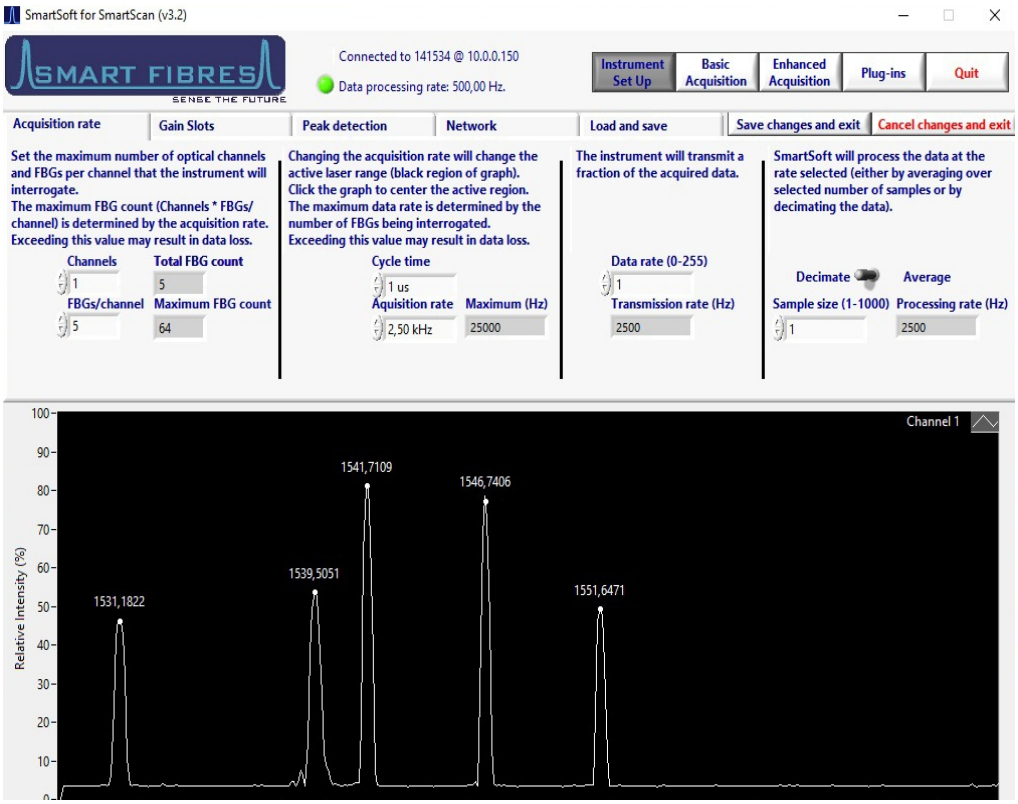

Figure 3.5. SmartSoft Application Software.

To a more efficient and appropriate interaction with the interrogator, two libraries were created previously, one of which is responsible for reading data straight from the Ethernet connection. The library allows to extract two sorts of data peakData and rawData (discarded in this thesis).

3.4 Middleware

The middleware client is designed to run in a multi-thread context and handle continuous peak data at a high rate. The received packet must be processed into a new format that is compatible with the cloud platform database. Each measurement is linked to a unique set of metadata (position of the sensors on the object, time-stamp, variation, etc.). Metatada is essential since data in the cloud is used by several layers of the architecture. The AR/VR framework's objective is real-time visualization, while the analytical framework's purpose is offline analysis.

The purpose of this thesis is to improve the middleware created, this middleware can be hosted in a Raspberry Pi 3 Model B board. This improvement is done by creating a middleware that is independent of the type of interrogator used, as well as filtering the data received from the interrogator, saving only the useful and purposed data, and removing the ability to send data directly to the viewer through TCP-IP connection. Other sub-improvements comprise resolving some issues with the structures used to preserve the data (buffer, queue, etc.) and finding a way to reduce mongoDB server traffic, in other words, to interact with mongoDB once a certain amount of collected items has been reached. Rearchitecting the middleware's classes and enhancing the middleware folder hierarchy are two of the most significant enhancements made to the middleware.
Every point of improvement will be discussed in further detail later.

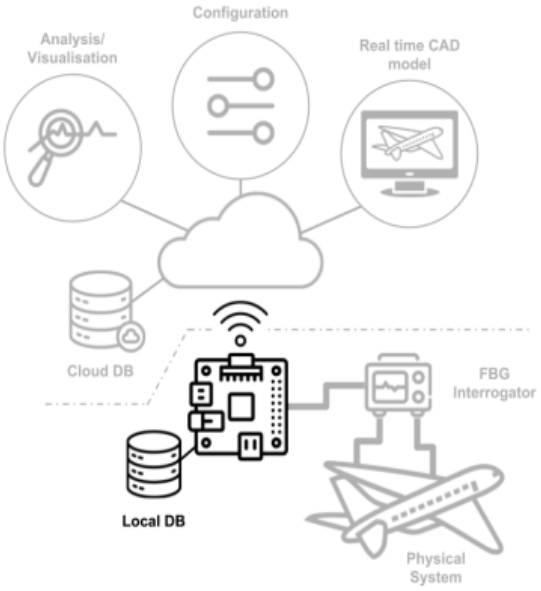

Figure 3.6. Middleware layer [1].

3.4.1 Middleware Classes Architecture

Before going into the architecture of the middleware classes, let's take a look at the various types of messages that the smartscan sends to the client.
Messages can be divided into four categories:

- Diagnostic messages
  These messages are used to set the interrogator's operational condition and to check its current status.
- Maintenance messages
  These messages are used to set the interrogator's configuration, such as transmission speed, scan speed, and so on.
- Scan data
  Raw data acquired by the interrogator.
- Continuous data
  The peak data stream detected by the interrogator.

Each sort of message is routed through a separate UDP socket. To read and parse data from the Smartscan interrogator, a C++ client library was created. It implements all of the abstract class's methods.

3.4.1.1 Old Middleware Architecture

The library has two methods: one that listens on the four UDP ports and another that parses the messages received. The run method of the abstract class is used to run these functions as independent threads. When a raw or peak data message arrives, the listen thread places it in a queue, while the parse thread pulls it from the queue, parses the data, and stores it in the appropriate vector. A custom C library implements the message queue, which uses a mutex to enable efficient thread-safe queue operations, while a semaphore synchronizes the parsing and listening threads.
The getPeakData and getRawData methods have been added to return to the caller a vector of

peakData or rawData structures. They're synchronized with the parsing thread, which releases them when a new UDP message is parsed. This allows all new data to be pushed to the remote database during the idle period between two following UDP transmissions. A keep alive thread function was also provided by the client library, which simply sends a diagnostic message at regular intervals to verify if the interrogator is available. [1].
The middleware includes the primary client process as well as three initiated threads, for a total of four threads running.

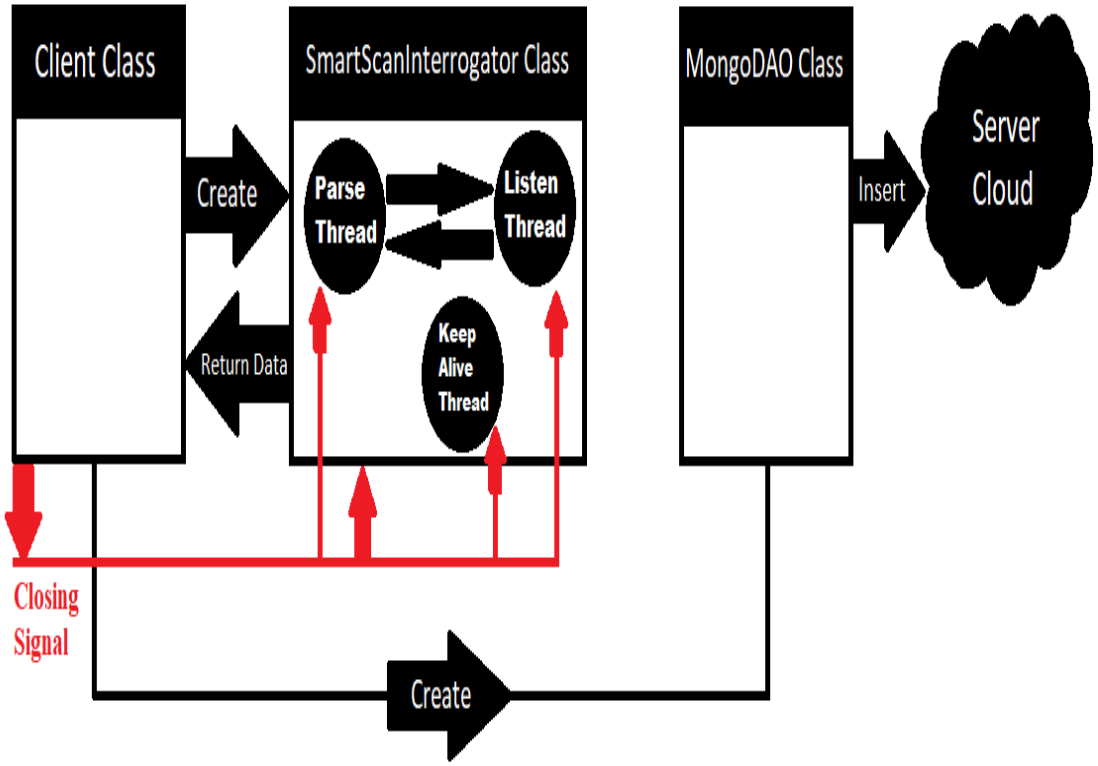

Figure 3.7. Old middleware architecture.

The closing signal sent to the client causes a break in the client class's while loop, as well as a break in all while loops in the smartscaninterrogator class, forcing the threads to join. The MongoDAO object is one of the objects that will be destroyed while the client is closing.

3.4.1.2 New Middleware Architecture

In this improvement, the middleware architecture is totally overhauled. As a result of the splitting, certain methods are also removed. The smartscaninterrogator class is split since it is responsible for listening, parsing, and storing data. By dividing the work of this class and its three threads, a new architectural approach is developed. The client will be initialized, create a listener object, and then be that listener object. The listener object will create a parser thread and wait for it to join in its destructor, and the parser thread will create a mongoDB insertion thread and wait for it to join in its destructor.
So sequential thread generation, sequential thread joining, and object destruction in this manner. In summary, the client will be the listener, and the listener will construct the parser thread, which

will then build the mongoDB insertion thread. When a closing signal is received, the parser thread waits for the mongoDB thread to join, and the listener waits for the parser thread to join. The client will close normally after thread joining is completed.

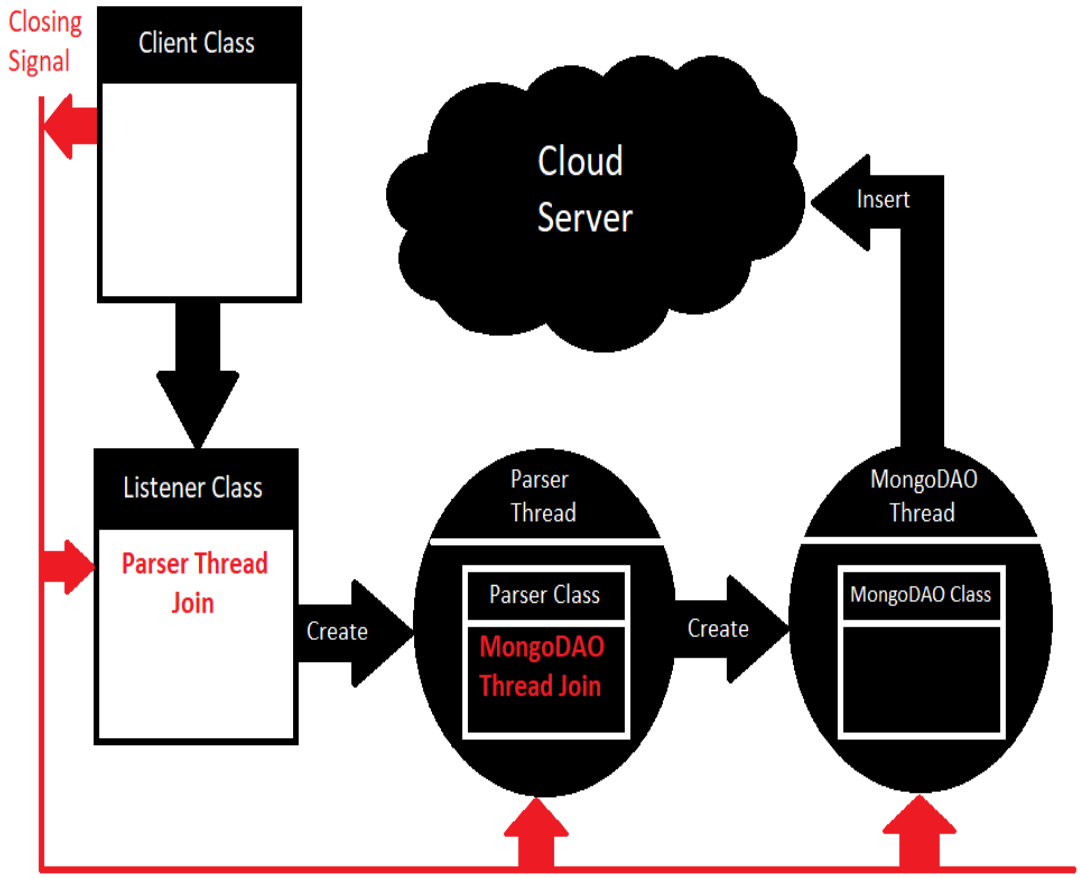

Figure 3.8. New middleware architecture.

3.4.2 Middleware Files Hierarchy

3.4.2.1 What is hierarchical file system ?

A hierarchical file system is how an operating system organizes and displays drives, folders, files, and other storage devices. The disks, folders, and files in a hierarchical file system are shown in groups, allowing the user to see the files they're interested in [28].

3.4.2.2 Old Middleware Files Hierarchy

The source files and headers were not well organized in the old middleware hierarchy, and a modules folder was present, containing the smartscaninterrogator source files, headers file, and its own cmake file. The middleware in this hierarchy was missing the two implemented libraries that are responsible for communicating with the interrogator and retrieving two sorts of data from the interrogator: peakData and rawData. These libraries had to be installed on the operating system before the header files could import them.

As a result of this disorganization, the hierarchy was not well arranged and was not clear to the user. By using the command "tree" from the terminal, figure 3.9 reveals the file hierarchy of the old middleware.

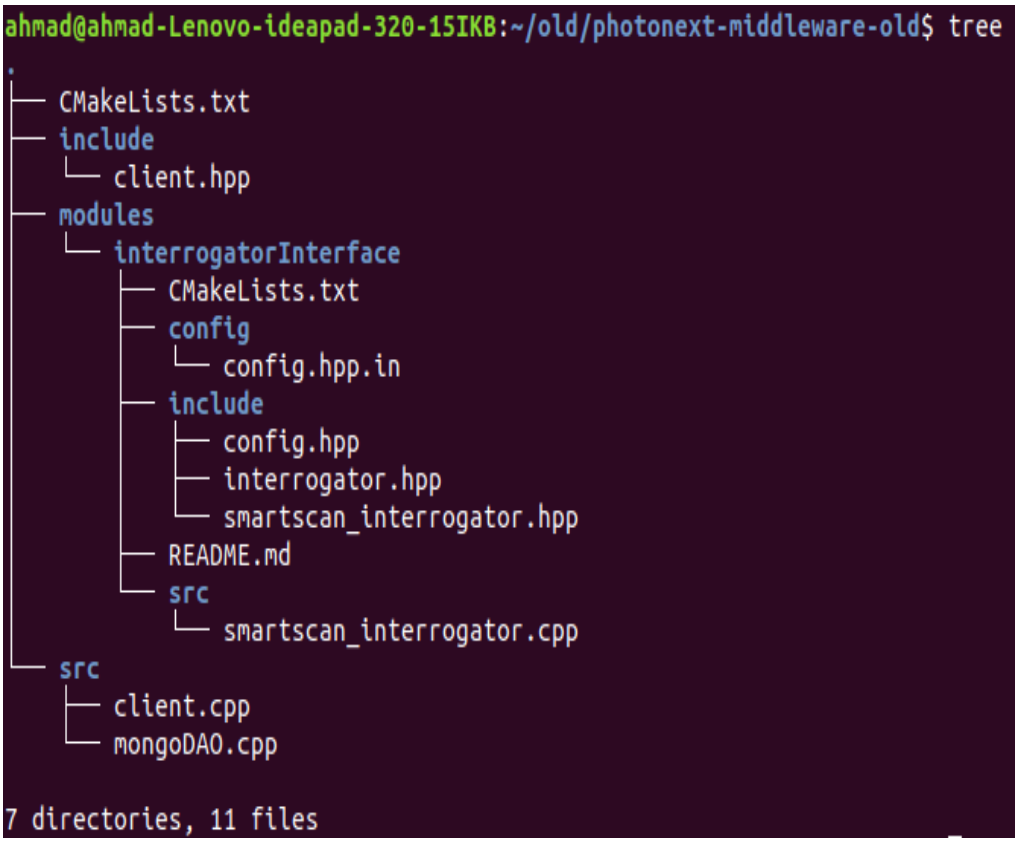

Figure 3.9. Old middleware files hierarchy.

3.4.2.3 New Middleware Files Hierarchy

The new middleware structure is well-organized in the sense that there will be an include file inside the main middleware folder that will contain the header files arranged into sub-folders. The CommonLibraries folder will contain the header files for the two libraries that are responsible for communicating with the interrogator and retrieving two types of data: peakData and rawData. The header files for the SmartScan interrogator source files will be found in SmartScanBlue folder. As a result, header files for each interrogator are split in this way.
Similarly the src folder, which will have sub-folders containing the appropriate source files. The source files for the two libraries required to communicate with the interrogator and get data will be found in the CommonLibraries folder, while the source files for the SmartScan interrogator will be found in SmartScanBlue.
There is no need to install libraries into the operating system because all of the required libraries, from their header files to their source files, are contained within the middleware project folder. Following the hierarchy update, both source and header files are updated in parallel to import the required libraries with their new relative location in the middleware main folder.
As a result of this enhancement, there is no need to install libraries on the system, and middleware directories are better structured and more easily accessible by the user.
By using the command "tree" from the terminal, figure 3.10 reveals the file heirarchy of the new middleware.

```
ahmad@ahmad-Lenovo-ideapad-320-15IKB:~/middlewareFin/photonext-middleware$ tree
.
├── CMakeLists.txt
├── config
│   ├── config.hpp.in
│   └── smartscan_config.txt
├── include
│   ├── CommonLibraries
│   │   ├── libsmartscan
│   │   │   ├── data_queue.h
│   │   │   ├── msg_queue.h
│   │   │   ├── smartscan_constants.h
│   │   │   ├── smartscan_utils.h
│   │   │   └── socket.h
│   │   └── libutils
│   │       └── utils.h
│   └── SmartScanBlue
│       ├── client.hpp
│       ├── config.hpp
│       ├── interrogator.hpp
│       ├── listener.hpp
│       └── parser.hpp
└── src
    ├── CommonLibraries
    │   ├── libsmartscan
    │   │   ├── data_queue.c
    │   │   ├── msg_queue.c
    │   │   ├── smartscan_utils.c
    │   │   └── socket.c
    │   └── libutils
    │       └── utils.c
    └── SmartScanBlue
        ├── client.cpp
        ├── listener.cpp
        ├── mongoDAO.cpp
        └── parser.cpp

11 directories, 23 files
```

Figure 3.10. New middleware files hierarchy.

3.4.3 Middleware CMake Files

3.4.3.1 About CMake

CMake is an extensible, open-source system that manages the build process in an operating system and in a compiler-independent manner. Unlike many cross-platform systems, CMake is designed to be used in conjunction with the native build environment. Simple configuration files placed in each source directory (called CMakeLists.txt files) are used to generate standard build files (e.g., makefiles on Unix and projects/workspaces in Windows MSVC) which are used in the usual way. CMake can generate a native build environment that will compile source code, create libraries, generate wrappers and build executables in arbitrary combinations. CMake supports in-place and out-of-place builds, and can therefore support multiple builds from a single source tree. CMake also supports static and dynamic library builds. CMake is designed to support complex directory hierarchies and applications dependent on several libraries [29].

3.4.3.2 Old Middleware CMake Files

A main CMakeLists.txt file was present in the old middleware that would recure into all folders and build any detected other CMakeLists.txt file. To make its build, the file was written in such a way that it searches for installed libraries on the system, in addition to some included libraries. The libraries were linked to the target project if they were installed on the system. As an example, a piece of code is provided below.

```
1  find_library(LIBUTILS utils)
2  find_library(LIBSMARTSCAN smartscan) #smartscan utils library
3
4  if ( NOT LIBUTILS )
5    message(FATAL_ERROR "utils library not found")
6  endif()
7  if(NOT LIBSMARTSCAN)
8    message(FATAL_ERROR "smartscan library not found")
9  endif()
10
11 include_directories(include
12   ${CMAKE_CURRENT_SOURCE_DIR}/src
13 )
14 # Create a list with all .cpp source files
15 set( project_sources src/client.cpp src/mongoDAO.cpp)
16
17 add_executable(client ${project_sources})
18 target_link_libraries(client -lutils)
19 target_link_libraries(client -lsmartscan)
```

Figure 3.11. Piece of code of the old middleware CmakeLists.txt file.

3.4.3.3 New Middleware CMake Files

There is only one cmakefile in the new middleware. This file creates libraries by including all of their header and source files, then linking them to the target project.

```
1  add_library(SmartScan
2              src/CommonLibraries/libsmartscan/data_queue.c
3              src/CommonLibraries/libsmartscan/msg_queue.c
4              src/CommonLibraries/libsmartscan/smartscan_utils.c
5              src/CommonLibraries/libsmartscan/socket.c
6   )
7
8  target_include_directories (SmartScan PUBLIC
       ${CMAKE_CURRENT_SOURCE_DIR}/include/CommonLibraries/libsmartscan)
9
10 add_library(Utils
11             src/CommonLibraries/libutils/utils.c
12  )
13 target_include_directories (Utils PUBLIC
       ${CMAKE_CURRENT_SOURCE_DIR}/include/CommonLibraries/libutils)
14
15 add_executable(client ${project_sources})
16 target_link_libraries(client SmartScan)
17 target_link_libraries(client Utils)
```

Figure 3.12. Piece of code of the new middleware CmakeLists.txt file.

3.4.4 Middleware Client Class Modifications

3.4.4.1 Old Middleware

The old middleware's client class had the tcp/ip connection with the viewer; all of these methods were discarded and removed, as were all variables related to this connection.
SmartscanInterrogator and mongoDAO objects were created using the class's main function. The second task was to either have the peak data parsed and saved so that it could be entered into the mongoDB server, or to open a tcp/ip connection with the Viewer if the "-noDB" command line argument was inserted. If the case was a mongoDB insertion, three threads, listening, parsing, and keeping alive, were initialized. In the case of a Viewer TCP/IP connection, two threads were initialized, one for connecting with the viewer and the other for sending data it.
CTRL+C is the keyboard shortcut that can be used to close the client. Following the use of this keyboard shortcut, an interrupt signal is generated and handled by a signal handler function, which updates some variables that causes the return of the client main method.

```
1  //server tcp function
2  void server_tcp();
3
4  //client data thread
5  void send_tcp_datapacket();
6
7  //client connection
8  void send_tcp_packet(void*, uint32_t, bool);
9
10 //signal handling
11 void sigint_handler_mongo(int);
12 void sigint_handler(int);
13
14 //set up data structure for config packet
15 void set_up_confg(vector<bool> &);
16
17 //main function
18 int main(int, char **);
```

Figure 3.13. Functions used by the old middleware Client class.

3.4.4.2 New Middleware

The client will notice the name of the interrogator that will be inserted, and then it will proceed according to the name of this interrogator. An error will occur if the interrogator name is incorrect or if the interrogator name is missing. The listener object is created at the beginning, then initialized and prepared to listen the data arrival. To put it another way, the client will be the listener. The client will also be in charge of naming the collection that will be inserted into the MongoDB server. The name of the collection is set as a global variable since it will be used by other objects when inserting data into MongoDB, particularly by the mongoDAO object.
In addition to the main function, the new client has only two functions: one for naming the collection that will be created on the mongoDB server, and the other for signal handling.
CTRL+C and CTRL+Z are the keyboard shortcuts that can used to close the client. Following the use of these keyboard shortcuts, an interrupt signal is generated and handled by a signal handler function, which updates specifically the atomic boolean variables i_running and i_connected causing a sequential joining of the executing threads.

```
1  //creating collection name
2  string create_collname(std::string);
3
4  //signal handling
5  void closing_client(int);
6
7  //main function
8  int main(int, char **);
```

Figure 3.14. Functions used by the new middleware Client class.

A piece of code of the new main function will be provided below.

```
1   if (argc == 2)
2      {
3          /**********************************
4          * Transform the argv to uppercase. *
5          **********************************/
6          std::string str = argv[1];
7          transform(str.begin(), str.end(), str.begin(), ::toupper);
8
9          /**************************************
10         * Blue Box Interrogator: "SMARTSCAN". *
11         **************************************/
12         if(str=="SMARTSCAN")
13         {
14             signal(SIGINT, closing_client);
15             signal(SIGTSTP, closing_client);
16             collectionName = create_collname(str);
17             ssi.i_init();
18             ssi.i_running.store(true);
19             ssi.ssi_listen();
20         }
21
22         /*********************************************
23         * Single Board Interrogator: "SMARTSCANSBI". *
24         *********************************************/
25         else if(str=="SMARTSCANSBI")
26         {
27
28         }
```

Figure 3.15. A piece of code of the main function of the new middleware Client class.

3.4.4.3 Conclusion

The main class is less coded in this new version, and the client is now interrogator independent, as the main function may handle any name or type of interrogator supplied.

3.4.5 Middleware SmartScanInterrogator Class Modifications

3.4.5.1 Old Middleware

The interrogator interface is implemented by the SmartScanInterrogator class, which overrides its functions.

```
1  //initializing the connection with the interrogator
2  int i_init();
3
4  //initializing the threads
5  void i_run();
6
7  //joining the thread
8  void i_stop();
9
10 //destroying mutex and semaphores created
11 void i_close();
12
13 //retrieving the rawData
14 void i_get_spectrum(vector<spectrumData> &);
15
16 //setting the speed for retrieving rawData
17 void i_set_raw_data_speed(int);
18
19 //retrieving peakData
20 void i_get_peak_data(vector<peakData> &);
21
22 //setting the speed for retrieving peakData
23 void i_set_peak_data_speed(int);
24
25 //setting the scan speed
26 void i_set_scan_speed(int,int,int);
27
28 //socket methods
29 int ssi_socket_init();
30 void ssi_socket_delete();
31
32 //thread functions
33 void ssi_parse_message();
34 void ssi_listen();
35 void ssi_keepalive();
36
37 //parse methods
38 int ssi_parse_diagnostic_msg(uint8_t*, size_t, uint8_t *);
39 int ssi_parse_maintenance_msg(uint8_t*, size_t, SSI_CONFIG *);
40 int ssi_parse_cont_data_msg(uint8_t*, size_t);
41 int ssi_parse_scan_data_msg(uint8_t*, size_t);
42
43 //send message methods
44 int ssi_send_maintenance_msg();
45 int ssi_send_timestamp();
```

Figure 3.16. Functions used by the old middleware SmartScanInterrogator class.

The class will initialize all locks, mutexes, semaphores, queues, and the log file at the beginning. After all of the interrogator configuration has been concluded, socket initialization will take place. The i_init() function handles all these steps.
The listen_th thread, which will handle the ssi_listen() function, the parse_th thread, which will handle the ssi_parse_message() function, and the keepalive_th thread, which will handle the ssi_keepalive() function, are all created in the i_run() function. The ssi_listen() function will begin listing on the socket that was previously initialized with the interrogator; each socket handles a specific type of message. Diagnostic messages, maintenance messages, scan data, and continuous data are the four types of messages that will arrive, as previously stated. The listener thread will then start filling the shared msg_q queue with the parser function. When msg_q is ready for the parser to read it, the listener will notify it with a semaphore sem_post(&lock_s) that the parser is already waiting for it using sem_timedwait(&lock_s,&wait_timeout_cont).
The parser thread will begin populating its queue (raw or peak data queue) based on the received message by invoking the appropriate parsing functions. then the client will use one of the following functions to get the data: i get spectrum or i get peak data; the data is then saved in a vector and inserted into the mongoDB server by the help of the mongoDAO object and its functions. When the client's execution loop is broken, it will call the i_stop() function, which will store a false value in the the atomic boolean variables i_running and i_connected and wait for the threads to join when their while loops have been broken.

3.4.5.2 New Middleware

The SmartScanInterrogator class is splitted into two classes: a Listener class and a Parser class. As a result, the divided classes will be less coded and include fewer functions, making them more readable to the user.
These two classes will be explained clearly, with their functions taken into account.

- **Listener**
  At the beginning, the class will initialize all locks, mutexes, semaphores, queues, and the log file. Socket initialization will proceed when all of the interrogator configuration has been completed. All of these procedures are handled by the i_init() function.
  The ssi_listen() function will create a parser object, with its constructor taking the reference of the listener synchronization variables, so that both classes' variables will have the same values on every update. The parser thread will then be initiated to handle the ssi_parse_message() function found in the parser class.
  The class will then begin listening in the same way as the old middleware, with one minor difference: the addition of a write file descriptor (fd_set write_fds). The reason for introducing this descriptor is that the old middleware had a problem connecting to the interrogator because it didn't have this file descriptor to initialize the writing from the interrogator side before reading could be done correctly. The atomic boolean variables will be set to false by the listener constructor, but will be set to true after the connection with the interrogator is established. Socket deletion and parser thread join will be done in the listener destructor. So, after the parser thread has completed its task, it will join the listener object's destruction. All used queues, mutex, and locks will be destroyed by the destructor itself.
  In addition, the listener will have two extra functions for configuration and synchronization. The first is ssi send maintenance msg(), which sends the initial configuration after the interrogator connection is established for the first time while the second is ssi_send_timestamp(), which is invoked during the initialization step of the i_init(). A new variable, "listenerInitializedWell", is introduced to the parser class. This variable is used to determine whether the initialization step has been completed correctly. So, when the listener is launched, the value of this variable is false, but after initialization, it will be set to true. This variable is checked to prevent the destruction of uninitialized semaphores, mutexes, and locks, avoiding dump core problems.
  Some functions are discarded or removed, but they will be indicated at the end of this section.
  The listener class overall functions will be listed below.

```
1  //initializing the connection with the interrogator
2  int i_init();
3
4  //socket methods
5  int ssi_socket_init();
6  void ssi_socket_delete();
7
8  //listening function
9  void ssi_listen();
10
11 //send message methods
12 int ssi_send_maintenance_msg();
13 int ssi_send_timestamp();
```

Figure 3.17. Functions used by the new middleware Listener class.

- **Parser**
  The listener object will create a parser object, which will be injected into the parser thread, that will handle the ssi_parse_message() function. A new mongo thread will be created at the start of this parse function, which will handle data insertion into the mongoDB server by executing the ssi_insert_peak data function. The ssi_parse_message() function will then begin retrieving the data that has already been saved in the msg_q queue, as well as determining the type of data in it, such as diagnostic, maintenance, or continuous data. One of the parsing functions will be called based on the message type.
  Reading frames will be processed according to the active channels and gratings inside these data parsing functions. This data will then be filtered based on non-zero wavelength values, and vectors will be filled in and inserted into the mongoDB server. When this vector achieves a certain size of collected data, a notification will be given to the condition variable, which is already waiting in the ssi_insert_peak data function, and then the insertion will be performed using the mongoDAO object. Locks and condition variables are used to keep these two threads synchronized. After insertion, this vector will be cleared to repeat the procedure until the client receives a close signal. The configuration of the interragator, which only includes the active gratings of the channels, and the filtered peak data will both be inserted into the mongoDB server.
  Insertion is accomplished by invoking two wrapper functions: one for configuration, which is run just once, and the other for peak data, which is executed once the vector reaches a specific size. When a closing signal is received and the atomic boolean variables are updated, the parser and mongo threads will join simultaneously. Before joining, the parser will perform a quick check on the collected peak data inside the vector, so if the collected data are less than the required size, the remaining data will be inserted to the mongoDB server as well, and this will be the final insertion. Following that, the mongo thread will join in the parser destructor, and the parser thread will join in the listener destructor, resulting in the client's total closure. This successive joining will ensure a perfectly synchronized working system. Together with the mongo thread join, log file closing will take place in the parser destructor. Following this closing and joining, a short outline of the total active sensors and the total saved total into the mongoDB server is outputed.
  As a result, the parser will include all functions responsible for parsing the received message and determining its type as well as analyzing and filtering them. After these procedures, the vector that will be injected into the wrapper function responsible for communicating with the mongoDB server will start to be filled up.

```
1  //Mongo Insertion Wrapper Functions
2  insert_config_mongo(mongodbDAO*,std::vector<bool>);
3  void insert_unity_data_mongo(mongodbDAO* ,int ,float ,uint64_t);
4  void insert_config_mongo_multiple(std::vector<bool>);
5  void insert_multiple_data_mongo(std::vector<cleanPeakData>);
6
7  //setting up the interrogator config
8  void set_up_confg(vector<bool> &);
9
10 //Parser main function
11 void ssi_parse_message();
12
13 //Mongo thread function
14 void ssi_insert_Peak_Data();
15
16 //Parsing Functions
17 int ssi_parse_diagnostic_msg(uint8_t*, size_t, uint8_t *);
18 int ssi_parse_maintenance_msg(uint8_t*, size_t, SSI_CONFIG *);
19 int ssi_parse_cont_data_msg(uint8_t*, size_t);
```

Figure 3.18. Functions used by the new middleware Parser class.

A new data structure is used by the parser, the name of the structure is cleanPeakData that will be used to store the collected, cleaned, and filtered data.

```
1  struct cleanPeakData
2  {
3      int index;
4      float wavelength;
5      uint64_t timestamp;
6  };
```

Figure 3.19. CleanPeakData structure.

3.4.6 Middleware MongoDAO Class Modifications

A Data Access Object (DAO) is a design pattern that provides an abstract interface to a database or other persistent storage system. The DAO supports some specific data operations without revealing database information by mapping application calls to the persistence layer.
To connect to MongoDB, this class is responsible for setting the Server, Database, User, and Password connection properties. MongoDB stores data in BSON format both internally, and over the network. Since MongoDB was used as the storage database, the application must be able to connect to it to execute CRUD operations. To accomplish this, the MongoDB community has released mongocxx, a library that provides basic database management functionality to C++ applications, which depend on the C libmongoc library.

3.4.6.1 Old Middleware

The mongoDAO object will handle the insertion after the connection with the mongoDB server has been established and the data has been prepared. When a single data is ready to be inserted, the

parser will invoke one of the insertUnityData() or the insertConfig() functions of the mongoDAO object, which will then connect to the required collection and construct the document using BSON with the values to be put into the server.

```
1 //Inserts peak/configuration data into the database
2 void insertUnityData(int, uint64_t, float, uint64_t);
3 void insertConfig(sensorConfig);
4
5 //Lists the databases available
6 void listDB();
7
8 //Prints all the documents in a selected collection
9 void showDB();
```

Figure 3.20. Functions used by the Old Middleware MongoDAO Class.

The insert_one() function is performed on the collection when the document is ready, thus inserting this document into the MongoDB server. In both insertUnityData() and insertConfig() functions, insert_one() is called.
The example code below shows how BSON prepares the document and how it is inserted in the specific collection.

```
1  void mongodbDAO::insertUnityData(int index, uint64_t timestamp, float
       wavelength, uint64_t curr_time){
2
3      mongocxx::collection coll = db.collection(collectionName);
4      long ts = timestamp;
5      long ct = curr_time;
6      auto builder = bsoncxx::builder::stream::document{};
7      bsoncxx::document::value doc_value = builder
8          << "type" << "peakData"
9          << "curr_time" << (int64_t) ct
10         << "index" << index
11         << "timestamp" << (int64_t) ts
12         << "wavelength" << wavelength
13         << bsoncxx::builder::stream::finalize;
14
15     bsoncxx::stdx::optional<mongocxx::result::insert_one> result =
           coll.insert_one(doc_value.view());
16 }
```

Figure 3.21. MongoDAO's InsertUnityData() function.

3.4.6.2 New Middleware

MongoDAO will only have two functions: one for the configurations insertion and the other for the peak data insertion. The insertion is done in such a way that the mongoDAO object will not take a single data to insert it, but rather a vector of data, and then bson will prepare the documents inside the insert functions. BSON will start documenting the peak data vector data by data after which every document will be pushed into a document vector created by BSON.

When all documents are pushed inside the vector and no more peak data to be read, the database will call the insert many function to insert all of the documents inside this vector in one shot.

```
1  //Multiple peak data insertion into the database
2  void mongodbDAO::insertMultipleData(std::vector<cleanPeakData>, uint64_t);
3
4  //Multiple configuration data insertion into the database
5  void mongodbDAO::insertConfigMultiple(std::vector<sensorConfig>)
```

Figure 3.22. Functions used by the New Middleware MongoDAO Class.

The insert_many() function is performed on the collection when all documents are ready, thus inserting these documents into the MongoDB server. In both insertMultipleData() and insertConfigMultiple() functions, insert_many() is called.
The example code below shows how BSON prepares the documents and how it is inserted in the specific collection.

```
1  void mongodbDAO::insertMultipleData(std::vector<cleanPeakData> data, uint64_t
       curr_time)
2  {
3
4      mongocxx::collection coll = db.collection(collectionName);
5      cleanPeakData data_local;
6
7      std::vector<bsoncxx::document::value> documents;
8      while (!data.empty())
9      {
10         data_local = data.back();
11         data.pop_back();
12
13         documents.push_back(
14             bsoncxx::builder::stream::document{}
15             << "type" << "peakData"
16             << "curr_time" << (int64_t)curr_time
17             << "index" << data_local.index
18             << "timestamp" << (int64_t)data_local.timestamp
19             << "wavelength" << data_local.wavelength
20             << finalize);
21
22     }
23
24     bsoncxx::stdx::optional<mongocxx::result::insert_many> result =
           coll.insert_many(documents);
25
26 }
```

Figure 3.23. MongoDAO's insertMultipleData() function.

3.4.7 Middleware Data Models

The library provides the rawData and peakData data models, which are the two data models that derive from sensors. The middleware decorates these data into Unity Peak Data and stores it in the database. Configuration data, which is used to set up the viewer's sensors, is another sort of data stored in the database.

3.4.7.1 RawData

Raw data are the raw information coming from the interrogator. Raw data is encoded on a 2 bytes integer. Raw data was discarded in this thesis for reasons that will be explained later.

```
1  typedef struct rawData
2  {
3      uint32_t rd_timestamp_sc;
4      uint32_t rd_timestamp_fr;
5
6      uint16_t rd_data;
7      uint16_t rd_slot;
8
9  } rawData;
```

Figure 3.24. RawData structure.

- **uint32_t rd_timestamp_sc**
  Represents the current raw data's seconds in UNIX Epoch time.
- **uint32_t rd_timestamp_fr**
  Represents the current raw data's fractions of second in UNIX Epoch time.
- **uint16_t rd_data**
  Represents the value of the sensor data in its raw state.
- **uint16_t rd_slot**
  Represents the raw data's metadata.

3.4.7.2 PeakData

The channel and grating indexes can be used to identify each peak value, which refers to a distinct sensor within the fibre link. Interrogators commonly have multiple channels to connect multiple optical fiber lines at once.

- **uint32_t pd_timestamp_sc**
  Represents the current peak data's seconds in UNIX Epoch time.
- **uint32_t pd_timestamp_fr**
  Represents the current peak data's fractions of second in UNIX Epoch time.
- **uint16_t pd_data**
  Represents the value of the simulation's peak data.
- **double pd_wavelength**
  Represents the current peak data's wavelength value.

- **uint8_t pd_channel**
  Represents the current peak data channel.
- **uint8_t pd_grating**
  Represents the grating of the FBG's current channel

```
1  typedef struct peakData
2  {
3      uint32_t pd_timestamp_sc;
4      uint32_t pd_timestamp_fr;
5
6      uint16_t pd_data;
7      double pd_wavelength;
8      uint8_t pd_channel;
9      uint8_t pd_grating;
10
11 } peakData;
```

Figure 3.25. PeakData structure.

3.4.7.3 CleanPeakData

It's the data structure that'll be injected in mongoDAO functions.

```
1  struct cleanPeakData
2  {
3      int index;
4      float wavelength;
5      uint64_t timestamp;
6  };
```

Figure 3.26. CleanPeakData structure.

The cleanpeakdata structure will be used to enter the document into the database, and two variables will be appended to it: one is a string, and the other is an uint64_t. As a result, the overall document model will include the following variables:

- **string type**
  Represents the document's data type. "peakData" or "config" are the only two options for the type.
- **unit64_t curr_time**
  Represents the UNIX Epoch timestamp (in seconds and fractions) at the moment that the data was inserted. This data can be used to calculate the application's elaboration time.
- **int index**
  Represents the current sensor's index. Since the system can hold 64 FBG sensors, it is calculated as $(channel * 16 + grating)$.
- **uint64_t timestamp**
  Represents the timestamp derived from the peak data detection.
- **double wavelength**
  Represents the wavelength of the sensor derived from peakData.

3.4.7.4 Configuration Data

The configuration of the FBGs in the physical system is represented by this data. The interrogator can handle 64 FBGs split across four channels, each with 16 gratings. 3.27, is designed for interrogators that can give more information about the sensors. Unfortunately, the only information retrieved from the library is the peakData of each sensor.

```
1  struct vec3 {
2      float x, y, z;
3      vec3(float _x, float _y, float _z) : x(_x), y(_y), z(_z) {};
4  };
5
6  struct sensorConfig {
7      uint8_t channel, grating;
8      bool is_active;
9      vec3 position;
10     float wavelength_idle;
11     float wavelength_var;
12
13     sensorConfig(uint8_t c, uint8_t g, bool ia, vec3 pos, float wi, float wv):
14         channel(c), grating(g), is_active(ia), position(pos),
15         wavelength_idle(wi), wavelength_var(wv) {};
16 };
```

Figure 3.27. Configuration data structure.

- **uint8_t channel**
  Represents the current peak data channel. The system can have a maximum of four channels.
- **uint8_t grating**
  Represents the grating of the FBG's current channel. Each channel in the system can have up to 16 gratings.
- **bool is_active**
  Represents a boolean indicating whether or not the present sensor (channel and grating) is active. The FBG is active if the value is "true," otherwise "false."
- **vec3 position**
  Represents the sensor's tridimensional position on the 3D representation of an airplane. At the time of acquisition, this data is filled with (0,0,0);
- **float wavelength_idle**
  Represents the sensor wavelength's idle value. Unfortunately, this value cannot be obtained from the interrogator; instead, it must be calculated in the Viewer.
- **float wavelength_var**
  Represents the sensor's wavelength variance. Unfortunately, this value cannot be obtained from the interrogator; instead, it must be calculated in the Viewer.

3.4.7.5 Document Data Model

This data model is a BSON document that represents the data injected into the database. It can handle peak data as well as configuration data.

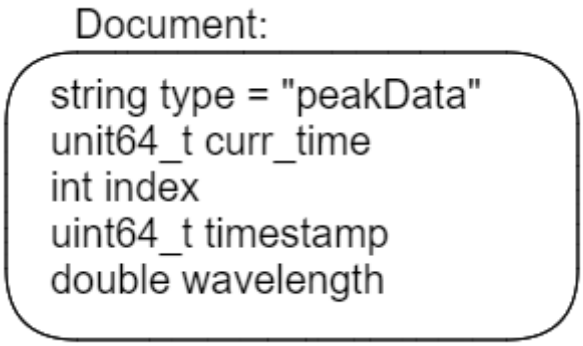

Figure 3.28. CleanPeakData data document model.

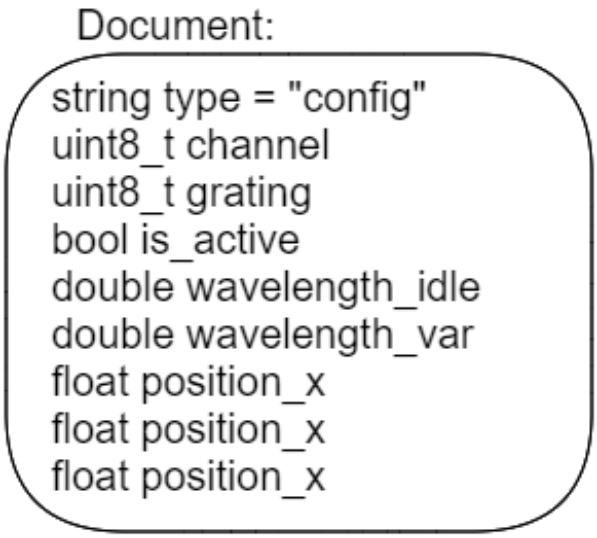

Figure 3.29. Configuration data document model.

3.4.7.6 Why Raw Data Is Discarded

The main reason for discarding raw data is that peak data is derived from it using a proper algorithm, while the second reason is because strain and temperature measurements are performed using the Brag-wavelength rather than directly from the intensity of the reflected signal. The intensity of the reflected peak is used to test the efficiency of the FBG network; in other words, a significant decrease in the amplitude of the intensity indicates that the FBG junction creates attenuation of the signal, causing a decrease in the signal to be equal to noise.

3.5 MongoDB Cloud

MongoDB is an open source NoSQL database management program. MongoDB is a tool that can manage document-oriented information, store or retrieve information.
MongoDB can handle a wide range of data types.MongoDB's design is made up of collections and documents rather than tables and rows, like in relational databases. MongoDB's basic data unit is the document. The documents resemble JavaScript Object Notation, although they employ a binary JSON variation (BSON). The advantage of utilizing BSON is that it can handle a

wider range of data formats. Like other NoSQL databases, MongoDB doesn't require predefined schemas. It stores any type of data. This allows users to create any amount of fields in a document, allowing MongoDB databases scale more easily than relational databases.
The basic methods of interacting with a MongoDB server are called CRUD operations. CRUD stands for Create, Read, Update, and Delete. These CRUD methods are the primary ways to manage the data in databases. This sub-chapter will define CRUD operations and explain how to execute them in MongoDB using the MongoDB Query Language (MQL).
A brief CRUD definition will be provided before taking a look at how to manipulate data with MongoDB CRUD methods.

3.5.1 What Is CRUD In MongoDB?

The user-interface patterns that allow users to view, search, and alter sections of a database are referred to as CRUD operations. Connecting to a server, querying the appropriate documents, and then changing the setting properties before sending the data back to the database to be changed are all ways that MongoDB documents can be modified. CRUD is data-driven and follows HTTP action verbs as a standard.
When it comes to the individual CRUD operations, they are as follows:

- **Create**
  Is used to insert new documents in the MongoDB database.
- **Read**
  Is used to query a database document.
- **Update**
  Is used to modify the contents of existing documents in the database.
- **Delete**
  Is used to remove documents from the database.

3.5.2 How To Perform CRUD Operations.

After that, the MongoDB CRUD operations are defined, and the next step is to explain how to carry out the individual operations and manipulate documents in a MongoDB database. The techniques of creating, reading, updating, and deleting documents will be detailed [33].

3.5.2.1 Create Operations

If the provided collection does not exist when MongoDB CRUD is executed, the create action will create it. MongoDB's create operations work on a single collection rather than several collections. Insert operations in MongoDB are atomic on a single document level. To add documents to a collection in MongoDB, there are two alternative creation operations available:

1. **db.collection.insertOne()**
   InsertOne(), as the title indicates, allows you to add one document to the collection. A new document is created if the create action succeeds. The function will return an object with the value "true" for "acknowledged" and the value "ObjectId" for "insertID".
2. **db.collection.insertMany()**
   It's possible to insert multiple items at one time by calling the insertMany() method on the desired collection. Multiple items are passed into the chosen collection and separated by commas in this example. Brackets are used within the parenthesis to indicate that a list of several entries is passed. This is known as a nested method.

```
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5fd989674e6b9ceb8665c57d")
}
```

Figure 3.30. Return object of insertOne() operation.

```
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("5fd98ea9ce6e8850d88270b4"),
                ObjectId("5fd98ea9ce6e8850d88270b5")
        ]
}
```

Figure 3.31. Return object of insertMany() operation.

3.5.2.2 Read Operations

Special query filters and criteria can be supplied with read operations to indicate which documents are desired. The MongoDB documentation contains more information on the available query filters. Query modifiers can also be used to adjust the number of results returned.
There are two ways to read documents from a collection in MongoDB:

1. **db.collection.find()**
   Simply use the find() function on the selected collection to get all of the documents from it. All records presently in the collection will be returned if the find() function is called without any arguments. The previously described filtering criteria are used to choose which results should be returned in order to get more specific with a read operation and identify a desired part of the data. Searching by value is one of the most prevalent techniques to filter the results: ***db.collection.find({"key":"value"})***

2. **db.collection.findOne()**
   Simply use the findOne() method on the specified collection to get one document that meets the search parameters. If the query returns many documents, this function returns the first one in natural order, which reflects the order of documents on the disk. The function returns null if no documents match the search criteria.
   The syntax for the function is as follows: ***db.collection.findOne({query}, {projection})***

3.5.2.3 Update Operations

Update operations, like create operations, are atomic at the document level and operate on a single collection. Filters and criteria are used in an update procedure to choose the documents that need to be updated. Updates to documents should be done carefully, as they are permanent and cannot be rolled back. This also applies to delete operations. There are three possible ways to update documents in MongoDB CRUD:

1. **db.collection.updateOne()**
   The updateOne() method is used on a chosen collection to update an existing record and modify a single document by passing two arguments to the method: an update filter and an update action. The update filter specifies which items should be updated, whereas the

update action specifies how those items should be updated. Use the "$set" key to specify the fields to be updated as a value, after passing in the update filter. The first record that matches the provided filter will be updated using this technique.
The syntax for the function is as follows:
db.collection.updateOne({filter}, {$set:{Key: "Value"}) item **db.collection.updateMany()**

```
1 {
2         "acknowledged" : true,
3         "matchedCount" : 1,
4         "modifiedCount" : 1
5 }
```

Figure 3.32. Return object of updateOne() operation.

UpdateMany() is similar to inserting multiple items in that it updates several items by passing in a list of items.The syntax for updating a multiple document is the same as for updating single documents.
The syntax for the function is as follows:
db.collection.updateMany({filter}, {$set:{Key: "Value"})

```
1 {
2         "acknowledged" : true,
3         "matchedCount" : 3,
4         "modifiedCount" : 3
5 }
```

Figure 3.33. Return object of updateMany() operation.

2. **db.collection.replaceOne()**
   The replaceOne() method replaces a single document in a collection. replaceOne() replaces the entire document, therefore fields from the old document that aren't present in the new will be lost.
   The syntax for the function is as follows:
   db.collection.replaceOne({Key: "value"},{Key:"Value"}

```
1 {
2         "acknowledged" : true,
3         "matchedCount" : 1,
4         "modifiedCount" : 1
5 }
```

Figure 3.34. Return object of replaceOne() operation.

3.5.2.4 Delete Operations

Delete operations, like update and create operations, operate on a single collection. For a single document, delete operations are also atomic. Filters and criteria can be used to delete operations

to determine which documents should be removed from a collection. The read operations use the same syntax as the filter options.
There are two ways to delete records from a collection in MongoDB:

1. **db.collection.deleteOne()**
   On the MongoDB server, deleteOne() is used to remove a document from a specified collection. The object to be deleted is specified using a filter criteria. The first record that fits the provided filter is deleted.
   The syntax for the function is as follows: ***db.collection.deleteOne({"key":"value"})***

```
1 {
2         "acknowledged" : true,
3         "deletedCount" : 1
4 }
```

Figure 3.35. Return object of deleteOne() operation.

2. **db.collection.deleteMany()**
   DeleteMany() is a method for deleting several documents from a specified collection in one operation. As in deleteOne, a list is supplied into the method, and the individual items are defined with filter criteria.
   The syntax for the function is as follows: ***db.collection.deleteMany({"key":"value"})***

```
1 {
2         "acknowledged" : true,
3         "deletedCount" : 3
4 }
```

Figure 3.36. Return object of deleteMany() operation.

3.6 Viewer

The Viewer's primary goal is to present data in a logical and intelligible manner, but the ultimate goal is to enable real-time and offline data analysis using multiple methodologies. The Viewer is a Desktop/AR application developed in Unity, that can read data directly from a MongoDB instance or from the middleware through TCP-IP, allowing it to work with a range of configurations.

Reading data directly from the middleware through TCP-IP connection is no longer possible with the upgraded middleware. As a result, the viewer can choose between two modes for interacting with the MongoDB server:

- **Real-Time**
  Using a MongoDB instance with Change Stream.
- **Non Real-Time**
  Using a MongoDB instance to read data that is already been recorded.

The application's home window is separated into two sections:

- **Import Model Menu**
  It allows to import a model of the device to be monitored in .obj format, as well as a default model of an airplane for simplicity.

- **Configuration Menu**
  There are four subareas in this section:

  - **Change HeatMap Colors**
    A collection of preset colors as well as a plus sign option to add additional.
  - **Server/Network Configuration**
    It has a drop-down menu with three different modes: TCP-IP, Real-Time, and Non-Real-Time, which can be used to configure the connection required to retrieve sensor information. The input form will allow to customize the connection based on the selection made in the drop-down menu.
  - **Measurement Configuration**
    It can be used to select the data type, such as temperature or displacement.
  - **Sensor Configuration**
    It can be used to get the sensor's configuration from the database or the middleware and change properties like their position on the model, idle time, and wavelength variability.

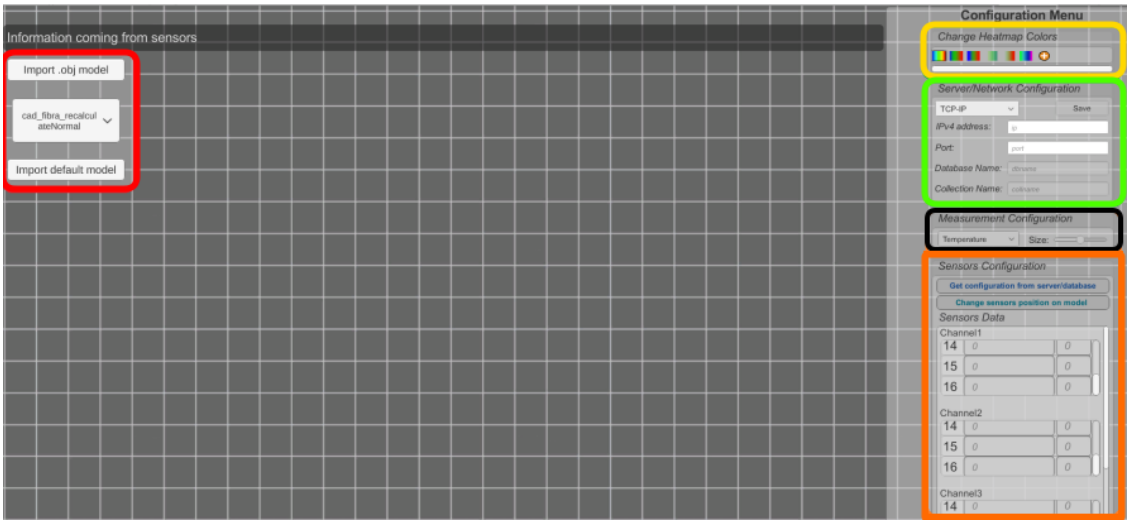

Figure 3.37. Home view of the Desktop application

3.6.1 Real-Time Mode

The Real-Time option, which makes use of MongoDB's ChangeStream capability, can be activated by selecting it from the drop-down menu in the Server/Network Configuration menu. The user can then configure the MongoDB instance's IPV4 address, port, database and collection name. The user can obtain the sensor configuration by clicking the {*Get configuration from server/database*} option from the Sensor Configuration menu. The last collection will be read because it will certainly be the one where the Middleware stores real-time data. The simulation view will be active and the menu will be removed when you press the {*Start Monitoring*} button. It will also begin reading from the ChangeStream feed.

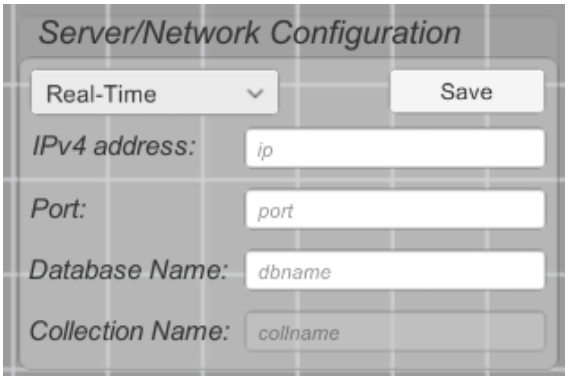

Figure 3.38. Real-Time Server/Network Configuration menu

3.6.2 Non-Real-Time Mode

The Non-Real-Time mode, which can be selected in the Server/Network Configuration drop-down menu, is an offline viewing of the sensors data from a previous simulation test. To retrieve the simulation's data, provide the IPv4 address, port, database and collection name in the appropriate sections. After pressing the Save button, the program will check the connection to the server; if it is successful, the user can receive the sensor configuration by pressing the {*Get configuration from server/database*} button. The peak data is read asynchronously from the database at the start of the simulation. Because of the poor performance of the selection query with a large amount of data, asynchronous programming is required [3].

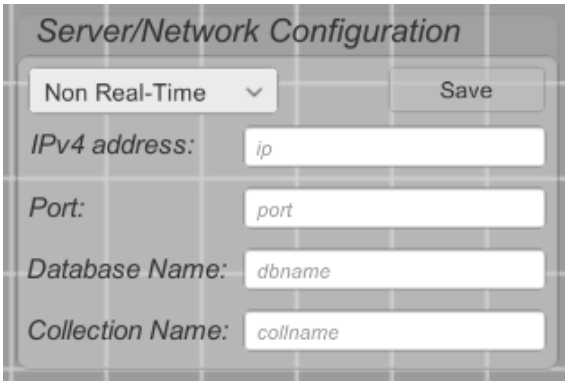

Figure 3.39. Non-Real-Time Server/Network Configuration menu.

3.6.3 Simulation's Outcome

The application will create a log file and a line graph at the end of each simulation as a summary of the simulation. The log file is a .csv file that contains all of the simulation's wavelengths at each timestamp.

```
Timestamp       ,Ch1Gr1   ,Ch1Gr2   ,Ch2Gr1
1613579105122918,1546.7190,1530.6010,1540.02
1613579106221705,1546.8470,1530.7100,1540.02
1613579107320490,1546.8490,1530.7100,1540.02
1613579108308992,1546.8500,1530.7100,1540.02
1613579109415673,1546.8520,1530.7100,1540.02
1613579110513960,1547.7320,1531.4490,1540.02
1613579111502955,1547.0360,1530.8280,1540.02
1613579112601740,1546.7640,1530.6180,1540.02
1613579113700121,1546.8750,1530.7080,1540.02
1613579114689108,1546.9040,1530.7180,1540.02
1613579115787395,1546.8890,1530.7070,1540.02
1613579116886164,1547.6050,1531.2710,1540.02
1613579117882964,1546.2450,1530.3170,1540.02
1613579118981345,1546.9080,1530.7100,1540.02
1613579119970254,1546.7320,1530.5870,1540.02
1613579121068634,1546.6300,1530.5440,1540.02
1613579122167418,1546.8800,1530.6920,1540.02
1613579123266286,1547.0170,1530.7830,1540.02
1613579124254782,1546.7660,1530.6110,1540.02
```

Figure 3.40. Example of a Log file with three active sensors [3] .

Following that, all wavelength data is written to a .csv file, the file is closed, and a Python script creates a line graph asynchronously from it.

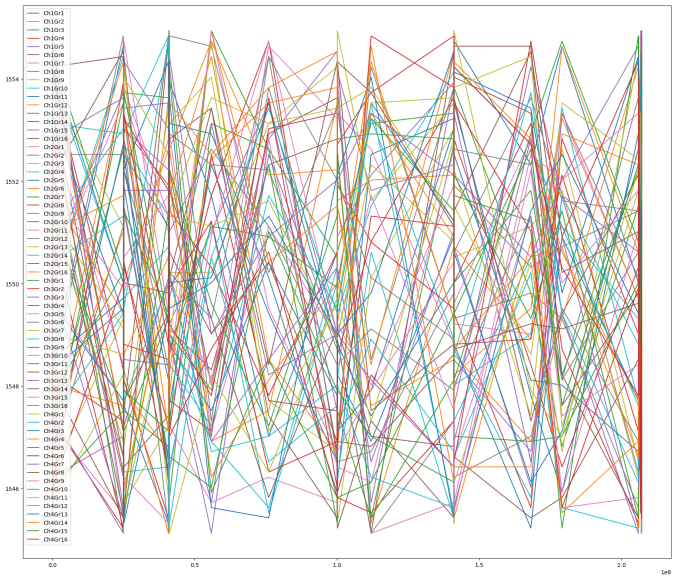

Figure 3.41. Example of a line graph image with 64 active sensors [3] .

Chapter 4

User Guide

4.1 MongoDB Compass

MongoDB Compass is the official MongoDB GUI, which is developed by MongoDB. MongoDB Compass assists users in making informed decisions about data structure, querying, indexing, and a variety of other database operations. The main goal of this chapter is to show MongoDB users how to utilize MongoDB Compass to perform database operations efficiently.

4.1.1 What Is MongoDB Compass?

MongoDB Compass is a graphical user interface for exploring, analyzing, and interacting with content in a MongoDB database without having to know or use queries. Compass is a graphical user interface for the Mongo shell.

4.1.2 What MongoDB Compass Can Do?

Compass can assist with a variety of tasks, such as data import and management, through an easy-to-use interface.

- **Importing data**
    1. Connect to a MongoDB deployment on MongoDB Atlas or a MongoDB deployment hosted locally on the machine.
       This method of importing will be discussed later.
    2. Import data into the MongoDB database from CSV or JSON files.
- **Querying data**
    1. Paste documents into the JSON view to add them to the collections, or use the field-by-field editor to manually add documents.
    2. Query data based on filtering the data with ad-hoc queries. Examine the collections for commonalities and trends.

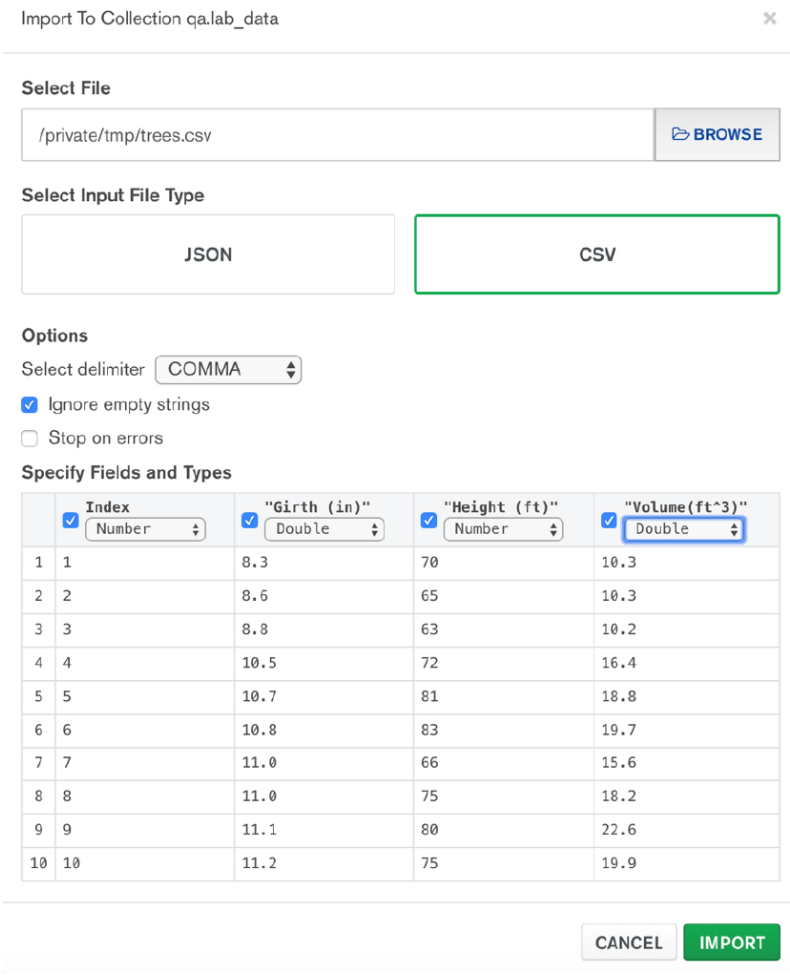

Figure 4.1. Importing data from CSV file [35].

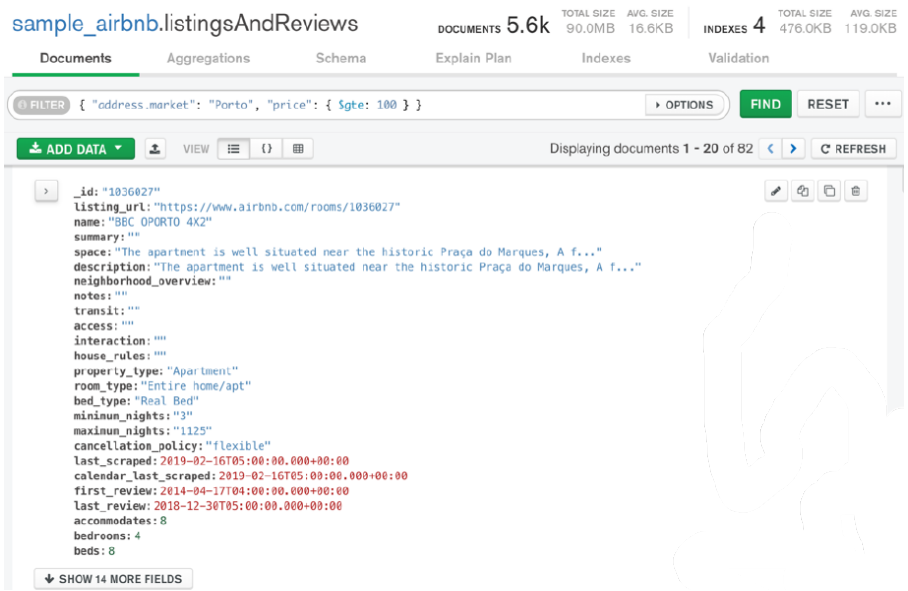

Figure 4.2. Querying on data [35].

- **Creating aggregation pipelines**

  1. Insert documents into the collections in two ways, JSON Mode and a Field-by-Field Editor.
  2. Write aggregation pipelines that allow documents in a collection or view to be processed into a set of aggregated results at multiple stages.

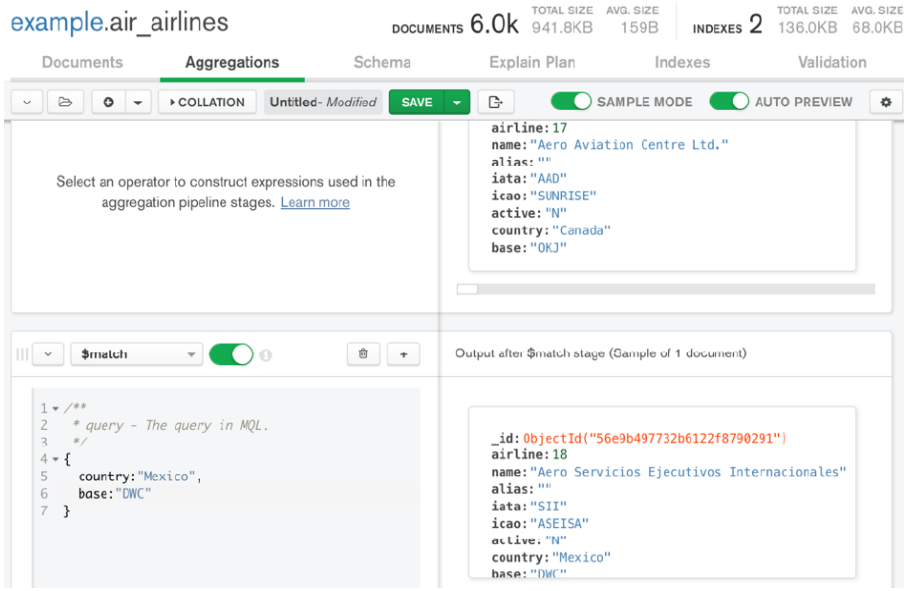

Figure 4.3. Creating aggregation pipelines [35].

- **Running commands in the shell**

  1. Connect to a MongoDB deployment on MongoDB Atlas or a MongoDB deployment hosted locally on the machine.
  2. Use Compass's inbuilt MongoDB Shell to interact with the data in a JavaScript context.

Figure 4.4. Running commands in a shell [35].

4.1.3 Connect To MongoDB Using Compass

When Compass is opened, an initial connection dialog appears. Compass offers two ways to connect to a deployment: either by supplying the deployment connection string or by filling in particular fields with the deployment information.

1. **Paste the connection string**
   Paste the connection string for the deployment directly into the dialog box if it is available, then click the Connect button to navigate to the Compass Home Page.

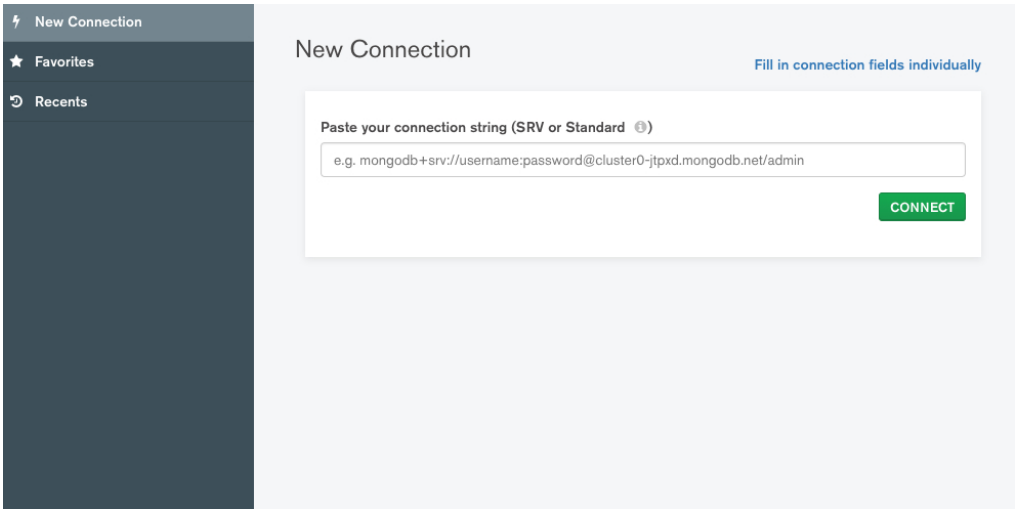

Figure 4.5. Pasting the connection string [35].

   To obtain the connection string for an Atlas cluster:

   (a) Navigate to the Atlas Clusters view.
   (b) Click Connect for the desired cluster.
   (c) Click Connect with MongoDB Compass.
   (d) Copy the provided connection string.

2. **Fill in Connection Fields Individually**
   Click Fill in connection fields individually to manually fill up specific connection fields.

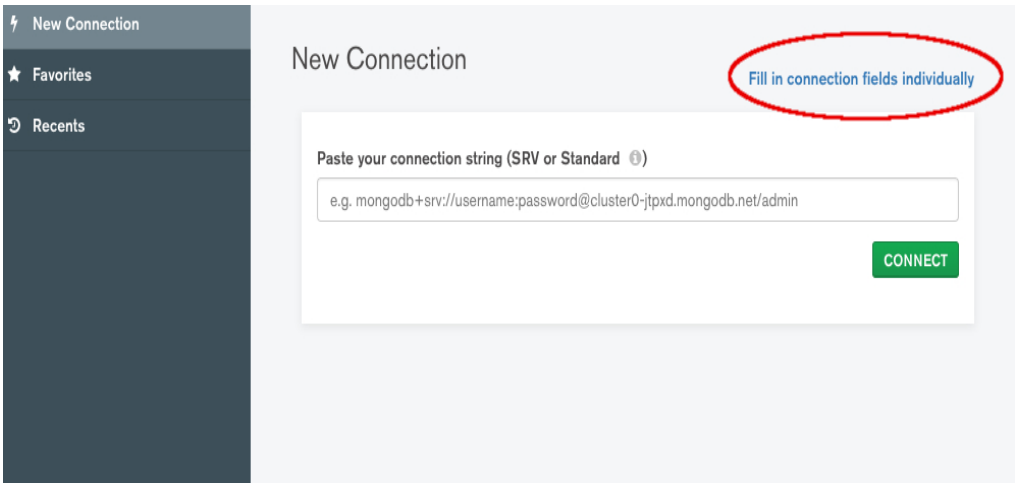

Figure 4.6. Filling connection fields [35].

A Hostname dialog will appear, with the connection field options as follows:

- **Hostname**
  The machine's hostname on which the deployment is running. This value is localhost if the deployment is running locally, and if it is connecting to an Atlas cluster, the hostname may be found in the Atlas cluster detail page.
- **Port**
  Port on which the deployment is running. If SRV Record is used to connect to the MongoDB deployment, it is not necessary. A standalone deployment uses port 27017 by default.
- **SRV Record**
  Indicates whether the provided Hostname is an SRV Record. If this toggle is enabled, no need to specify a port. SRV connection strings have a prefix of "mongodb+srv:". If SRV connection string is used, no need to include "mongodb+srv" in the Hostname.
- **Authentication**
  Authentication to use if the deployment requires authentication. Username/Password authentication is used in Atlas clusters.

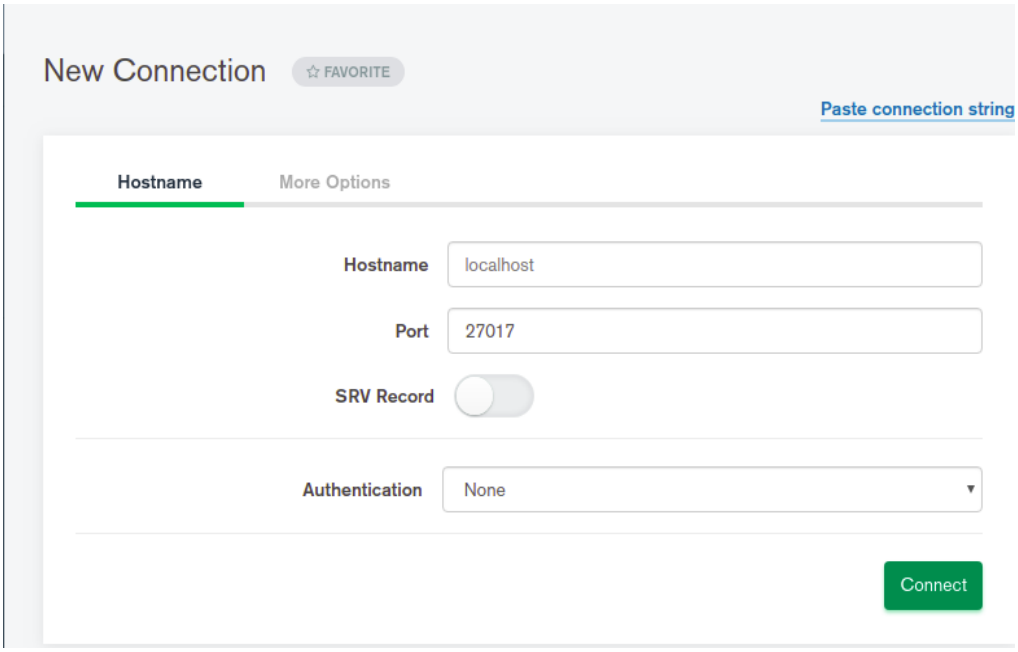

Figure 4.7. Hostname dialog fields.

4.1.4 Compass Home

The Compass Home screen displays information about the MongoDB instance that Compass is linked to, such as:

- The connection name if the connection is a favorite connection, or "My Cluster" otherwise.
- The type of deployment (standalone, replica set, sharded cluster). The number of replica set members will be displayed if the deployment is a replica set and the replica set name is supplied in the connection window.
- The MongoDB version, as well as the hostname and port.
- A list of the databases in the instance.

- Memory utilization, operation counts, and slowest operations are all included in the performance data.

The Compass Home screen can be accessed after connecting to a MongoDB instance by clicking the cluster name in the upper left corner.

The **Databases tab** on the home screen displays a list of all the databases connected to the current connection, along with their storage capacity, number of collections, and indexes.

The **Performance tab** displays real-time server performance information and graphs, including which database collections are the most often used, which operations take the most time to complete, and memory usage.

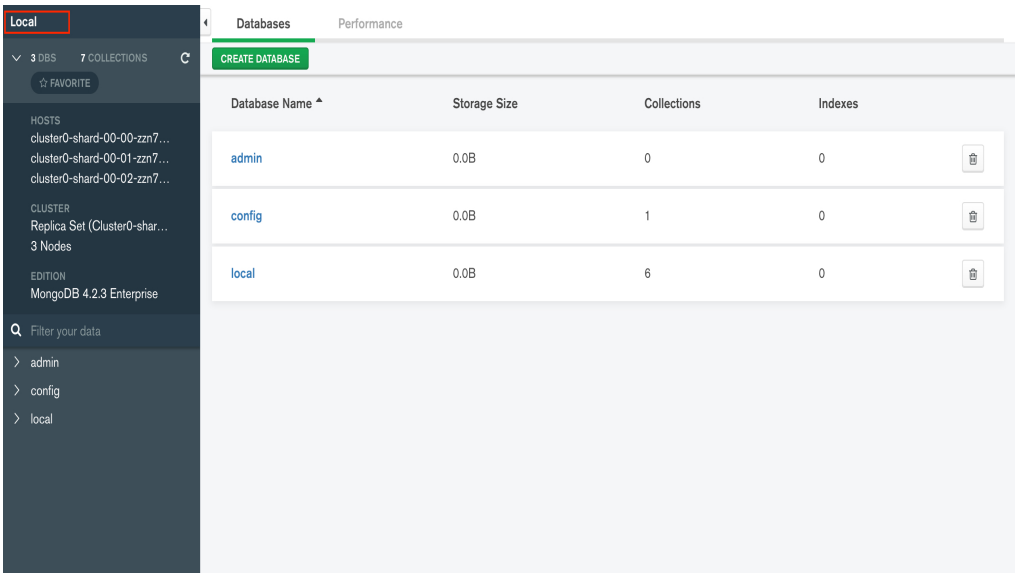

Figure 4.8. Compass home screen [35].

Chapter 5

Test And Evaluation

This chapter evaluates the proposed system's performance. This analysis will look into a variety of topics, but first it will examine the tools and equipment utilized in these topics.

5.1 Test Tools and Equipment

5.1.1 Sensors

In these tests, three different types of sensors are used.

- **Thermometric Probe**
  The thermometric probe is designed by DIMEAS engineers, it has 1 FBG grating.

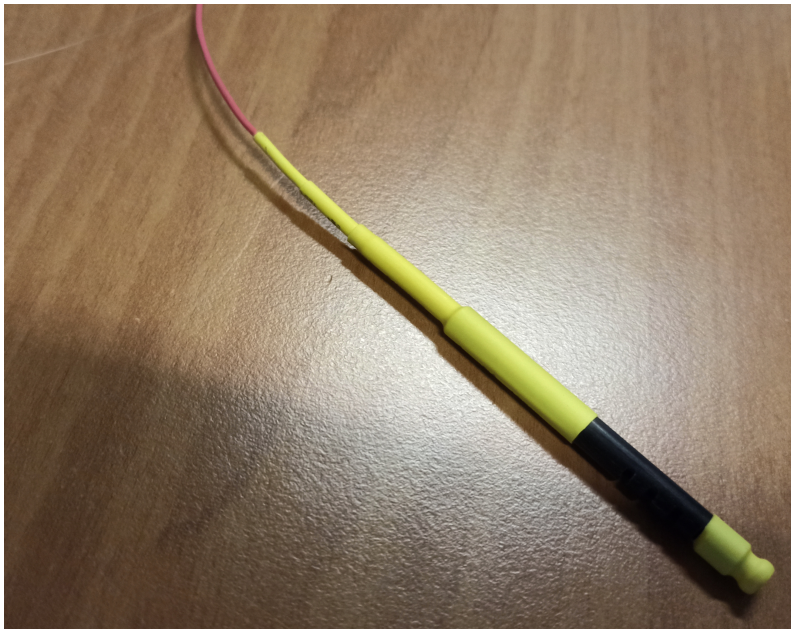

Figure 5.1. Thermometric probe.

- **Tension Bar**
  The bar has 1 FBG grating , it can be bent in two directions: forward and backward, with negative wavelengths achieved by backward bending and positive wavelengths achieved by forward bending.

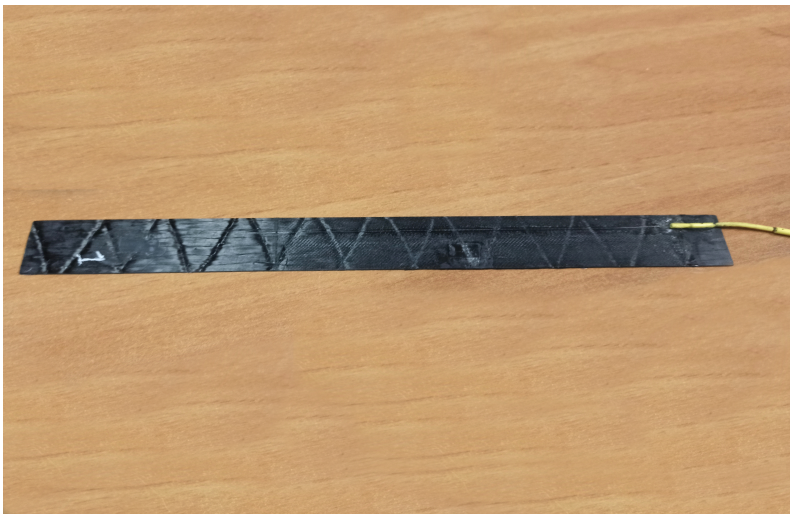

Figure 5.2. Tension bar.

- **ICARUS aircraft wing**
  The the aircraft wing's mechanism of movement is similar to that of tension bar, in that it can bend forward and backward. The aircraft wing has 5 FBG gratings integrated inside it.

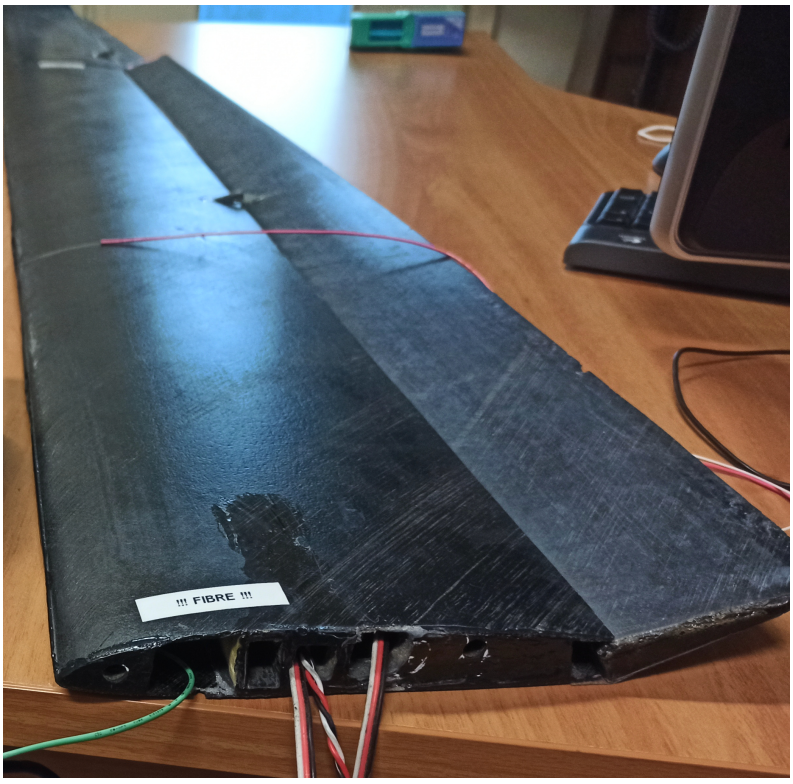

Figure 5.3. ICARUS aircraft wing.

5.1.2 SmartScan Interrogator

The tests are performed using the SmartScan© interrogator from SmartFibres©.

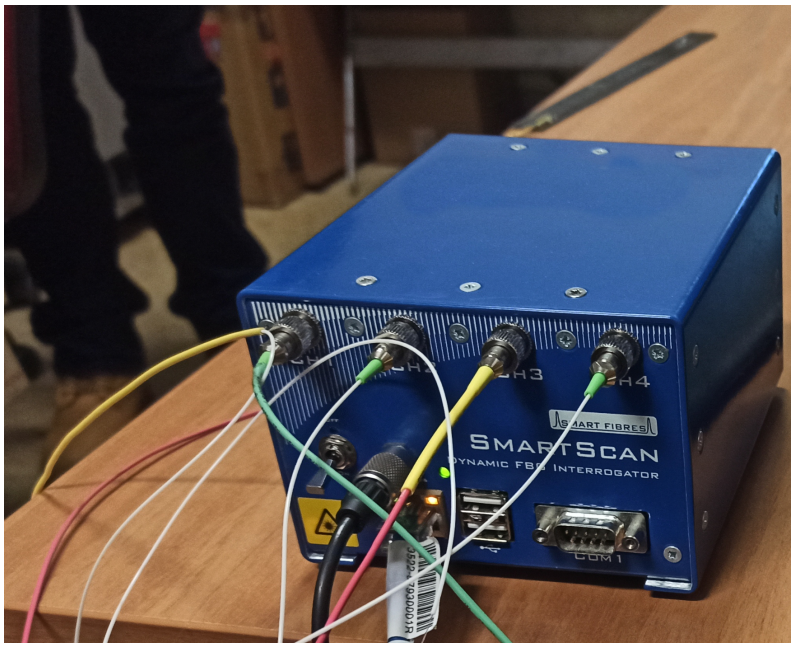

Figure 5.4. 4 active channels SmartScan© interrogator.

5.1.3 Raspberry Pi 3 Model B

The middleware will be hosted on the Raspberry Pi 3 Model B board, which will also be connected to the interrogator via Ethernet and to internet through a 4G USB adaptor. In figure below extra connections, such as a mouse, keyboard, and screen, are visible plugged to the board, which is powered by a power bank.

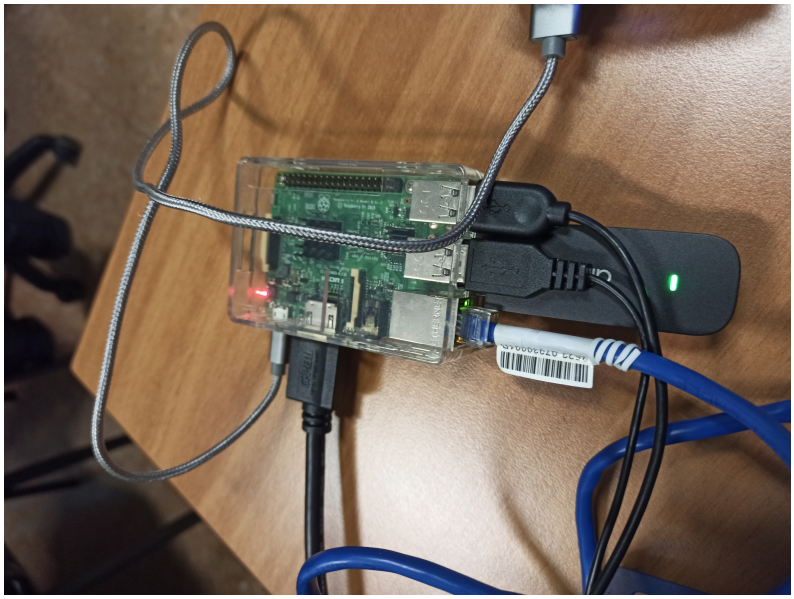

Figure 5.5. Raspberry Pi 3 Model B.

5.1.4 Portable Computer

The viewer and Mongodb Compass are installed on a Windows laptop that is powerful enough to not slow down the whole system execution.

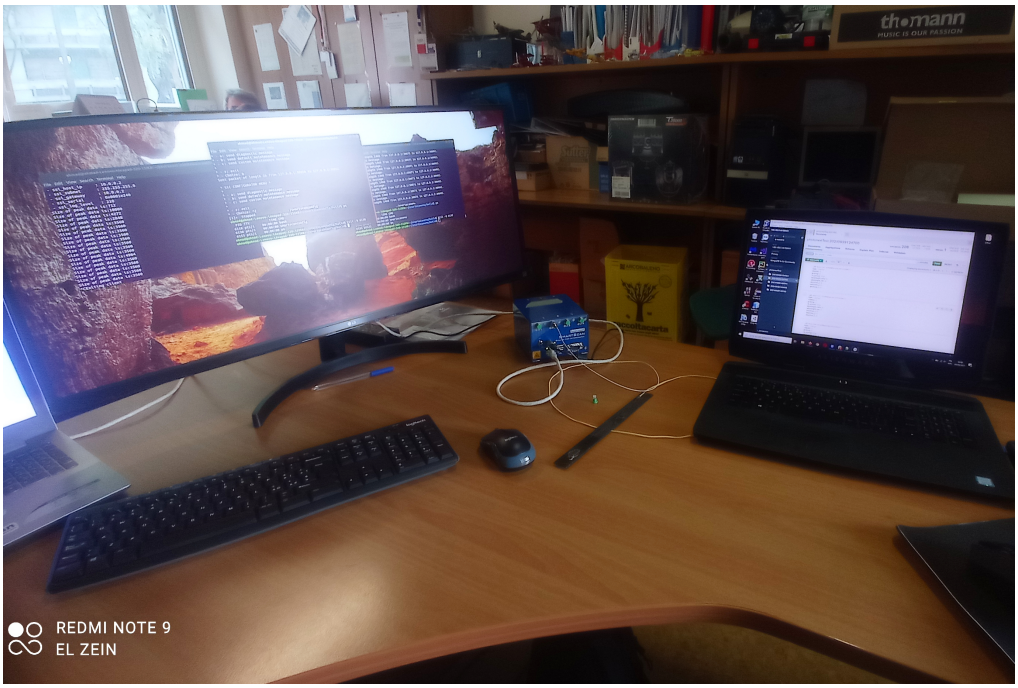

Figure 5.6. Portable computer launching the viewer and MongoDB Compass.

The overall delay from the middleware receiving the peak data to the visualization in the Desktop/AR application is used to compute the system latency. The program captures timestamps at each phase of the simulation, particularly in the interrogator, middleware, and viewer, during the flow.

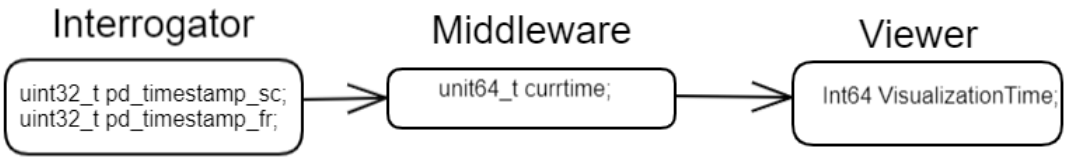

Figure 5.7. Timestamps stored during the simulation [3]

5.2 Test Scenarios

Both a SmartScan© interrogator emulator and a real physical interrogator are used in the tests. The emulator is a Linux application written in C that creates random UDP traffic in the same way as the SmartScan© interrogator.
The emulator will be configured to generate random signals on the first two gratings of each channel, giving us a total of eight active gratings, just like in the aircraft. While dealing with the real physical interrogator, all of the previously listed sensors will interfere in the test, and 8 gratings will be attached to the SmartScan© interrogator's 4 distinct channels in order to acquire the same gratings number as the airplane.

5.2.1 Connection Availability

5.2.1.1 Scenario

The test is to ensure that the middleware is properly connecting to the emulator or interrogator, as well as closing without failure. After several attempts and shifting from emulator to actual interrogator while considering the important variable "EMU_LOCAL" and initializing its value.

```
ahmad@ahmad-Lenovo-ideapad-320-15IKB:~/middlewareFin/photonext-middleware/build$ ./client smartscan
------------------------------------------------------------
*              SSI CONFIGURATION                           *
------------------------------------------------------------
 - ssi_demo        : 0
 - ssi_gratings    : 16
 - ssi_channels    : 4
 - ssi_raw_speed   : 0
 - ssi_cont_speed  : 25
 - ssi_scan_speed  : 400
 - ssi_first_fr    : 0
 - ssi_netif       : eth0
 - ssi_smsc_ip     : 10.0.0.150
 - ssi_host_ip     : 10.0.0.2
 - ssi_subnet      : 255.255.255.0
 - ssi_gateway     : 10.0.0.2
 - ssi_serial      : 0x0001e240
 - ssi_log_level   : 7
------------------------------------------------------------
*           STARTING DATA LISTENING                        *
------------------------------------------------------------
Collection Created On MongoDB Server:
SMARTSCAN_20211127173024163805222 4912
------------------------------------------------------------
Sending 20 Peak Data ...
Sending 20 Peak Data ...
Sending 20 Peak Data ...
```

Figure 5.8. Connection established between the interrogator and the middleware.

5.2.1.2 Conclusion

The conclusion that can be drawn is that the new middleware has no problems connecting with both the real interrogator and emulator as the old middleware did.

5.2.2 Middleware Memory Usage

5.2.2.1 Scenario

The purpose of the test is to see how much memory the middleware occupies after around an hour of execution. Memory consumption is measured using the linux command htop, and a comparison is done between the old and new middleware in this test. Before running either of the two version middlewares, htop command is used to check memory usage.

What is htop?
Htop command in Linux system is a command line utility that allows the user to interactively monitor the system's vital resources or server's processes in real time.

Following the htop command, a process name filter is performed. In the figure below, middleware is not launched since no client process is detected, and memory consumption is 176MB of the total Raspberry Pi 3 memory of 923MB, as shown.

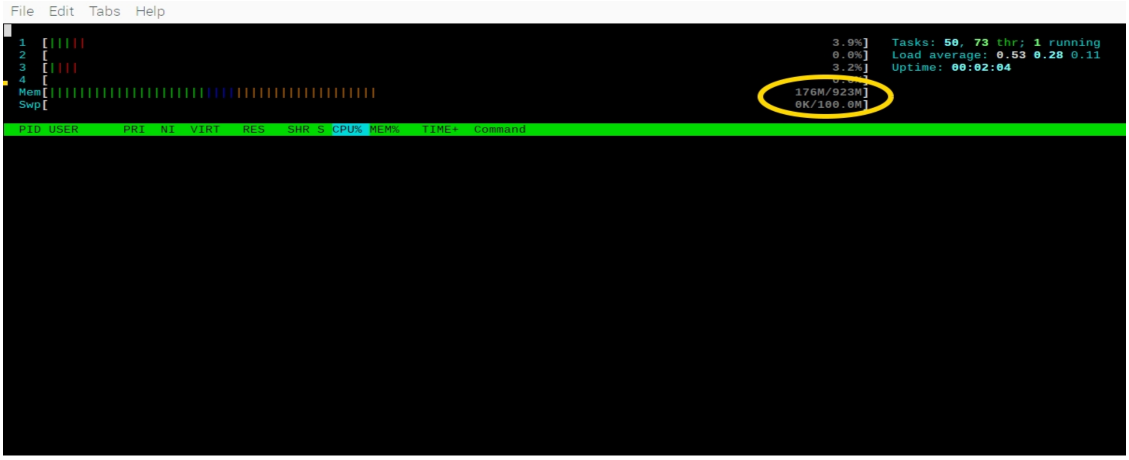

Figure 5.9. Raspberry Pi 3 Model B memory usage without launching the middleware.

After that, the old middleware is launched to check its memory consumption and utilization, and the consumption is calculated by detecting the total memory usage after launch and subtracting it from the startup memory usage.
The old middleware is shown in the figure below launching three threads with its main process, with a total memory usage of 470MB. So the old middleware memory consumption will be equal to 470M-176MB, which is equal to 294MB.

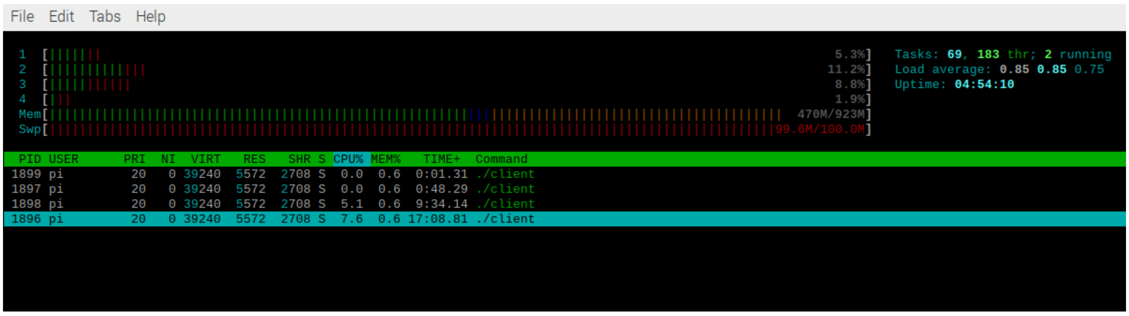

Figure 5.10. Raspberry Pi 3 Model B memory usage after launching the old middleware.

The new middleware undergoes the same memory test, and as shown in the figure below, the new middleware has two threads with its client main process, with a total memory usage of 255MB. As a result, the new middleware's memory consumption will be 255MB-176MB, which is equal to 79MB.

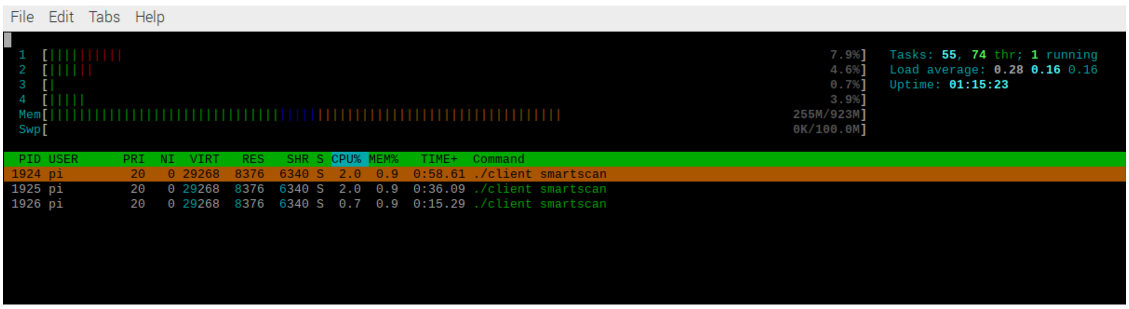

Figure 5.11. Raspberry Pi 3 Model B memory usage after launching the new middleware.

5.2.2.2 Conclusion

The conclusion is that the new middleware consumes less memory than the old middleware since it creates fewer threads and variables.

5.2.3 Data Rate And Middleware Stability

5.2.3.1 Scenario

The goal of the test is to verify the middleware's stability by running it for a few hours and comparing the total data stored in database to the total data sent and written on the standard output. After that, a data rate check is performed, which may be calculated by dividing the total data transmitted by the total time the middleware was executed. The middleware was executed for four hours and a half, and the total amount of data sent and stored in the MongoDB server was 887006.

- **Stability Confirmed**
  The middleware ran for four hours and a half without introducing any errors or failures, indicating that the middleware's stability is validated in this test.
- **Data Sent And Stored Equility Confirmed**
  After closing the client with CTRL-C or CTRL-Z, the total data sent will be written to the standard output, separated into two types of data: configuration data and peak data. As a result, after 4 hours and 30 minutes of execution, the total data sent is equal to 887006, which is sent to the standard output as shown in the figure below.

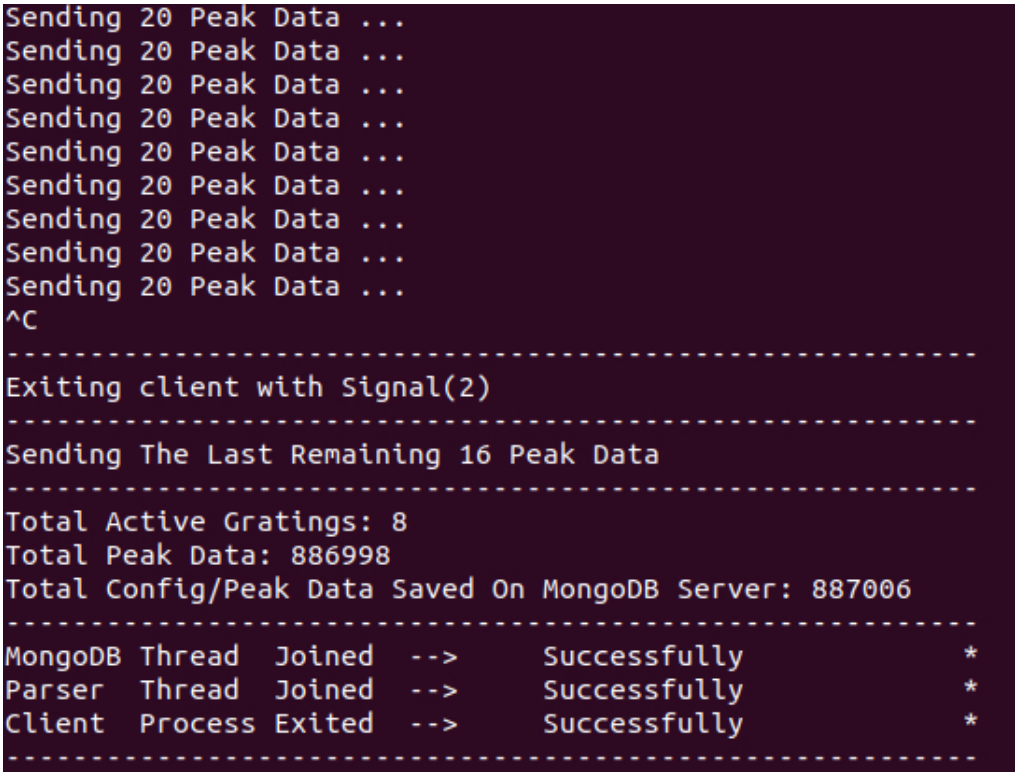

Figure 5.12. Total data sent in a 4 hours and a half period of time.

MongoDB Compass is used to check the stored data into the MongoDB database. The number of stored documents within the specific collection is checked, and as shown in the figure below, the number of saved documents is equal to 887006.

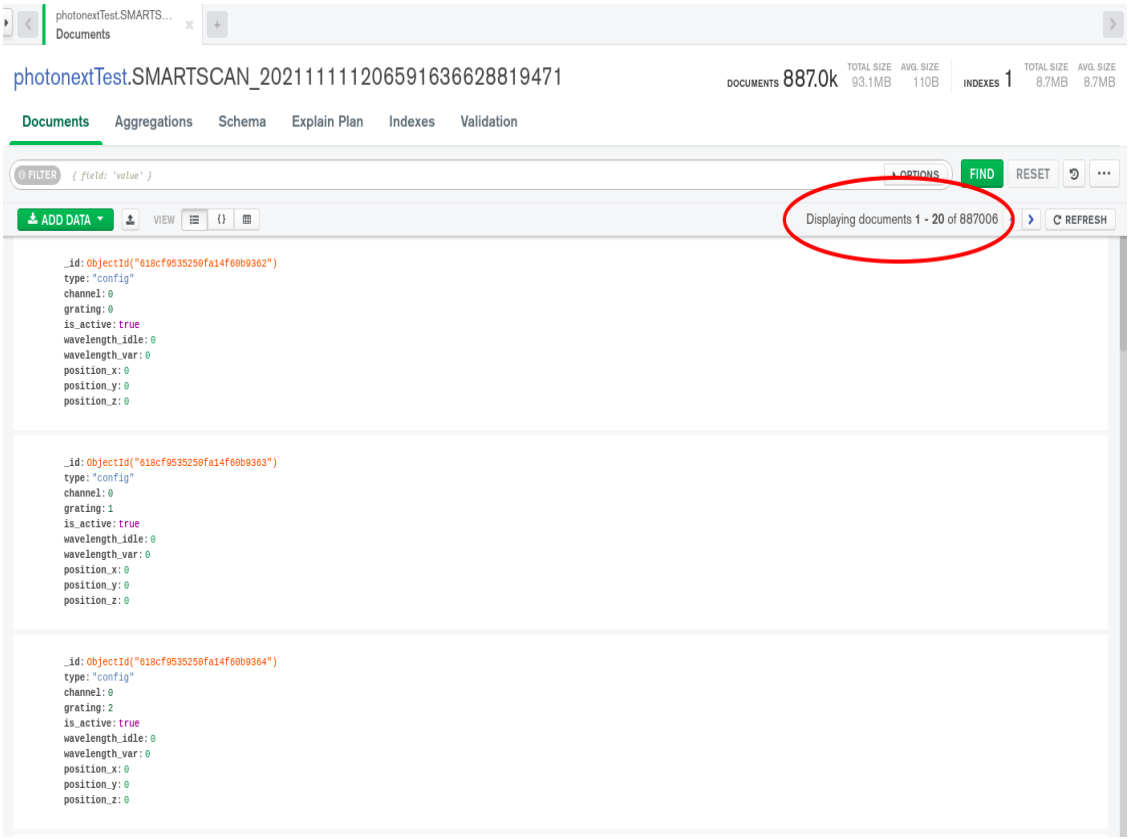

Figure 5.13. Total data stored in a 4 hours and a half period of time.

By comparing the data values in the two preceding figures, equality is verified; in other words, it is checked and certified that all data transmitted from the middleware to the MongoDB server is stored.

- **High Data Rate Confirmed**
  What is meant by data rate?
  The speed at which data is transferred within the computer or between a peripheral device and the computer, measured in bytes per second.

  In this test, data rate refers to the number of peak data sent per second by each active grating. In order to determine this data rate, some calculations will be performed, taking into account both old and new middleware. As previously stated, total data sent and stored in a period of four hours and a half was equal to 887006. Four and a half hours will equal 270 minutes, that is equal to 16200 seconds. Then, in 16200 seconds, 887006 data are transferred; to determine how much data is sent every second, divide 887006 by 16200, which equals around 55. If the total data sent from all active gratings is 55, divide that number by the total number of active gratings (8 active gratings) to get a close estimate which is equal to 7.
  As a result, the new middleware can collect 7 peak data each second for every grating.
  This is faster than the old middleware's collection rate, which was 2 data per second for each grating, taking in consideration that the total data sent/inserted by the old middleware in 4 hours and half was equal to 278100.

5.2.3.2 Conclusion

The conclusion that can be drawn is that the new middleware is a reliable program that does not suffer data loss and has a high data rate of 7 peak data per second for each grating.

5.2.4 Viewer Real Time Data Analysis

5.2.4.1 Scenario

In this test, the viewer will subscribe to the Change Stream's real-time feed, reacting in real time to the latest values entered into the MongoDB instance by the middleware.
The first goal of this test is to see if there is a delay while the viewer plots the wavelength variation graph of the newly entered data into the MongoDB instance, and the second goal is to see if the system returns to its initial working stability when the internet connection on the Raspberry Pi 3 is interrupted for a few seconds. After properly connecting all system components with 8 active gratings on the 4 channels of the interrogator, the middleware is launched on the Raspberry Pi 3 model B, and data begins to transit via the middleware on its way from the interrogator to the MongoDB server. The raspberry pi 3 model B is connected to the internet via a 4G USB adaptor; the reason for this is that the ICARUS aircraft will fly at a height of around 100 meters, which is still within the range of the 4G signal, thus it was chosen as the option to supply the board with internet connection. The viewer is keeping an eye on the MongoDB instance in real-time for any updates on the selected collection.
The figure below illustrates how all of the system's components are connected.

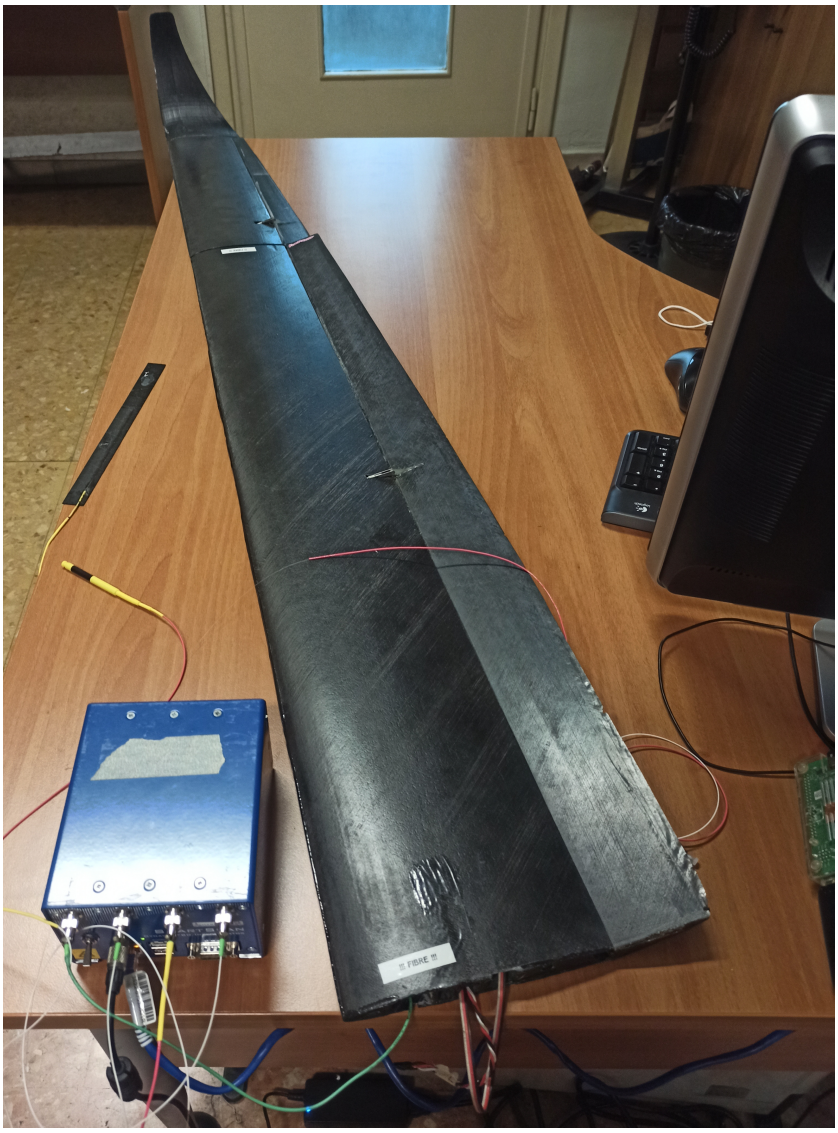

Figure 5.14. Full system under test.

- **No Network Interruption**

  While the system is running, any disturbance caused on one of the connected gratings, is directly recognized on the viewer graph, and the wavelength variation can be recognized without any delay, confirming that the viewer is operating in perfect real time mode with a delay of 1 or 2 seconds max depending on the connection of the portable computer on which the viewer is installed. However, based on repeated testing and the use of a mobile hotspot to provide an internet connection to the portable, the average latency can be considered negligible.

- **Network Interruption Occurs**

  Causing a network interrupt on the Raspberry Pi 3 means that the 4G USB adapter must remain plugged in but the 4G signal must vanish, effectively disconnecting the board from the internet and preventing the middleware from communicating with the MongoDB server and adding new data to the collection.
  Because the Raspberry Pi 3 has many cables attached to it, applying a Faraday cage to kill the signal coming to the 4G USB adaptor is difficult.

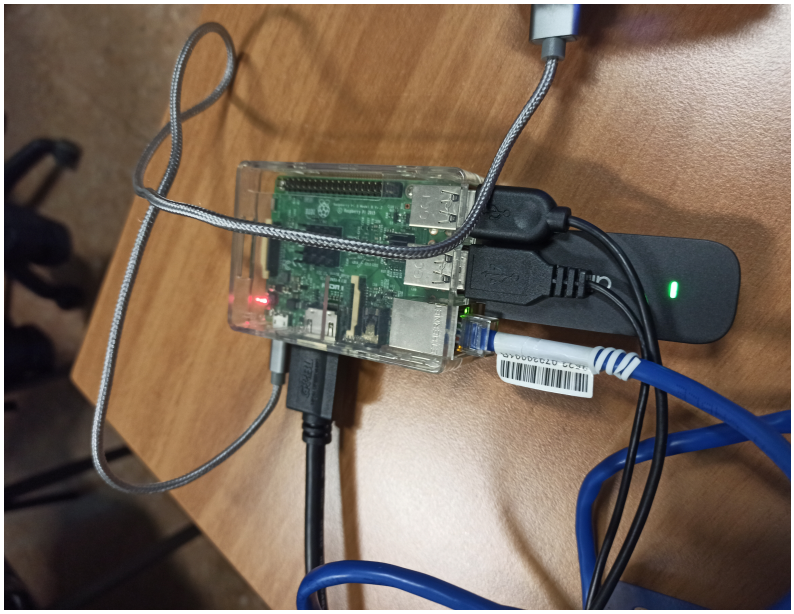

Figure 5.15. Raspberry Pi 3 Model B.

Another option is to use a mobile's USB tethering; in this case, the mobile will work similarly to a 4G USB adaptor, with the additional option of disabling mobile data; in this case, the network is not hardware disabled by unplugging the mobile, but is subject to an interruption by disabling and re-enabling mobile data. This method of interrupt is thought to be the closest to the one required, which is interrupting the signal on the 4G USB adaptor. The system will start, and then there will be a network interrupt, forcing the viewer graph lines for all active gratings to remain at 0, indicating that no new data is received and hence no wavelength variation. As a result, for all gratings activated, a straight line will remain around the zero values until the middleware reconnects to the MongoDB instance and new data is entered into the database, causing the graph lines to re-variate. After multiple tests, the middleware's reconnecting delay is averaged to reach a maximum value of 5 seconds. So, if the Raspberry Pi 3 encounters a network interruption, the middleware can reconnect in 5 seconds as a maximum delay.

The figure below shows a viewer graph of a test that the middleware starts inserting into the collection and immediately a network interrupt occurs, with the mobile data disabled for

around 30 seconds before being enabled. The viewer then begins reading the newly inserted data, and some gratings are subjected to some disturbance, as seen in the graph.

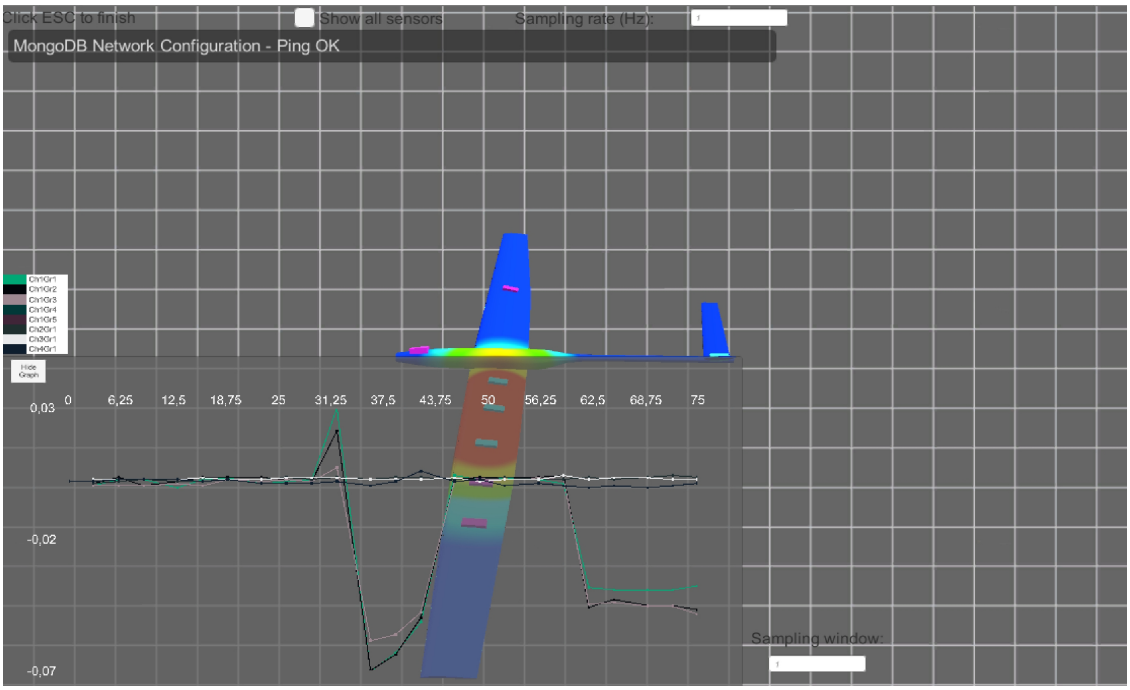

Figure 5.16. Model with 8 active gratings under real-time mode test.

5.2.4.2 Conclusion

The conclusion is that if the middleware has a network outage, it can reconnect and insert data into the MongoDB database in almost 5 seconds, after which the viewer will be re-updated with the new added data, causing the wavelength variation in its graph to be re-plotted.

Chapter 6

Conclusion

This concluding chapter will outline some suggestions for future development enhancements as well as conclusions on the work completed.

6.1 Future Work

This chapter outlines the planned platform's enhancements and new features that may be deployed in the near future. The project was built on the Git framework, which is the best choice for open source development and allows for bug management and feature proposals. It also improves collaboration by making it easier to track which modifications are done and making future work by other developers easier.

6.1.1 Missing Features

The project as a whole is running smoothly, but there are a few new features that need to be included. The most important feature that has yet to be built is the visualization and analysis framework, which will allow users to observe real-time data superimposed on the monitored item utilizing the AR/VR system. Taking into account that the Viewer (VR) system has already been developed and is now being enhanced and improved. While moving to the middleware, the type of interrogator is no longer a concern, and each new interrogator will demand the creation of new interrogator libraries as well as new classes with methods specialized to the interrogator.

6.1.1.1 AR framework

The goal of the Augmented Reality framework is to overlay data from the interrogator on top of the monitored physical system. You might be able to move around the object and watch how the data changes over time.
A smart wearable device, such as the Microsoft© HoloLens, acts as the AR layer.

Figure 6.1. Microsoft© HoloLens AR system [36].

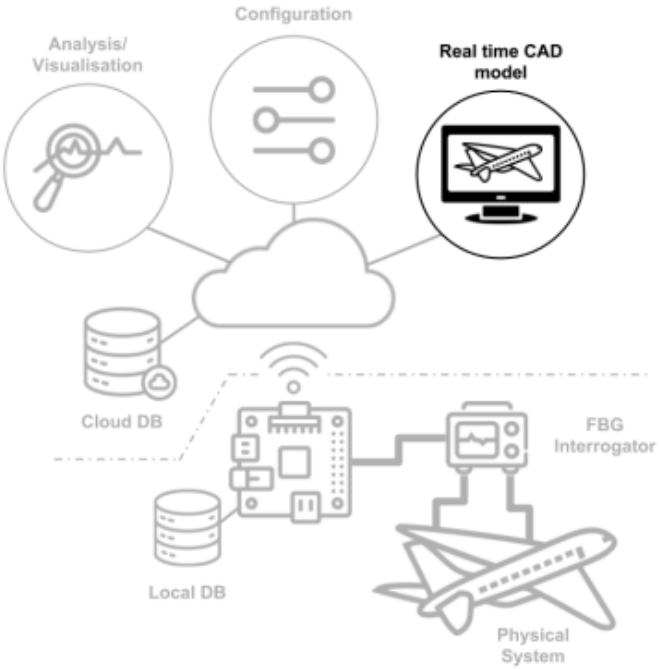

Figure 6.2. AR/VR visualisation and analysis [1].

6.2 Conclusion

The aim or goal of this graduation thesis (Improvement of a system for retrieving and displaying systematic aircraft data) is the improvement of PhotoNext project. The plan was to develop and implement a complete system that could retrieve data from any type of interrogator and display it in real time. The proposed middleware enhancement is a result of a thorough examination and analysis of the system's components.
This thesis concludes in the development of a reliable and stable middleware that is independent of the type of interrogator used, has a high data rate, requires less memory, and does not suffer from data loss or connecting issues.

Bibliography

[1] Mauro Guerrera. Algorithms and methods for fiber bragg gratings sensor networks. Master's thesis, Politecnico di Torino, 2018.

[2] Maria Giulia Canu. Mixed Real-Time Visualization Framework for FGB IoT sensors. Master's thesis, Politecnico di Torino, 2019.

[3] Antonio Scaldaferri. 3D visualization and analysis of a large amount of real-time and non-real-time data. Master's thesis, Politecnico di Torino, 2021.

[4] Gioele Baima. Design and Development of a Test Bench for Frequency Analysis of FBGs Optic Sensors for Prognostic Techniques for Aerospace Applications. Master's thesis, Politecnico di Torino, March 2019.

[5] Rod Stephens. Beginning Software Engineering. 2 March 2015. John Wiley & Sons. p. 94. DOI:10.1002/9781119209515.

[6] IBM Cloud Education. What is middleware?. 5 March 2021. IBM. https://www.ibm.com/cloud/learn/middleware.

[7] FBG Sensing System from SmartFibres©. https://www.smartfibres.com/technology.

[8] Patil,Gouri. A Seminar On Fiber Bragg Grating (FBG). Apr 2017. https://www.slideshare.net/Poornimagugale/fbg-ppt. PowerPoint Presentation.

[9] Mendoza Edgar A. Miniature Fiber Bragg Grating Sensor Interrogator(FBG-Transceiver™) System For Use in Aerospace and Automotive Health Monitoring Systems. Jan 2014, https://www.researchgate.net/publication/237665755_Miniature_fiber_Bragg_grating_sensor_interrogator_FBG-Transceiver_TM_system_for_use_in_aerospace_and_automotive_health_monitoring_systems_-_art_no_67580B.

[10] Bridget Botelho, Jack Vaughan. MongoDB. Tech Target. August 2020. https://searchdatamanagement.techtarget.com/definition/MongoDB.

[11] Madushanka, Tiroshan and Mendis, Laksheen and Liyanage, Dananji and Kumarasinghe, Chamath. Performance Comparison of NoSQL Databases in Pseudo Distributed Mode: Cassandra, MongoDB & Redis. 09 2015. Research Gate. https://www.researchgate.net/figure/Architecture-of-MongoDB_fig23_281629653.

[12] Priya Pedamkar. What is MongoDB. Educba. https://www.educba.com/what-is-mongodb/.

[13] What Is Real Time Application?. all about computer solutions. https://allaboutcomputersolutions.com/qa/what-is-real-time-application.html.

[14] Icarus - Polytechnic University of Turin. May 9. https://www.facebook.com/photo/?fbid=149491263854065&set=a.149491273854064.

[15] Infibra Technologies. FBG sensors overview. http://www.infibratechnologies.com/technologies/fiber-bragg-gratings.html.

[16] SmartScan© from SmartFibres©. https://www.smartfibres.com/products/smartscan.

[17] SmartSoft© from SmartFibres©. https://www.smartfibres.com/products/smartsoft-software.

[18] Rigoberto Jess. © Smart Fibres Ltd SmartScan FBG Interrogator. 2015. https://slideplayer.com/slide/3372740/. ppt presentation.

[19] Raspberry Pi 3. http://www.robo-dyne.com/shop/raspberry-pi-3/.

[20] Robert E. Hilpisch, Rob Duchscher, Mark Seel, Peter Soren, Kirk Hansen. Wireless communication protocol. 2004-12-01. https://patents.google.com/patent/US7529565B2/en.

[21] Ionos. TCP (Transmission Control Protocol) – The transmission protocol explained. 01.04.20. https://www.ionos.co.uk/digitalguide/server/know-how/introduction-to-tcp/.

[22] Lawrence Williams. What is the Difference Between TCP and UDP?. October 7, 2021. https://www.guru99.com/tcp-vs-udp-understanding-the-difference.html.

[23] Linda Rosencrance, George Lawton, Chuck Moozakis. User Datagram Protocol (UDP). October 2021. https://www.techtarget.com/searchnetworking/definition/UDP-User-Datagram-Protocol.
[24] Tushar Panhalkar. Introduction to TCP and UDP. https://info-savvy.com/introduction-to-tcp-and-udp/.
[25] Information Technology Services. Monolithic Application. 2007-09-02. https://en.wikipedia.org/wiki/Monolithic_application#cite_note-1.
[26] Siraj ul Haq. Introduction to Monolithic Architecture and MicroServices Architecture. May 2, 2018. https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63.
[27] STUDENTS@POLITO. The Politecnico in flight with the ICARUS team. 17 August 2017. https://poliflash.polito.it/en/students_polito/the_politecnico_in_flight_with_the_icarus_team.
[28] Computer Hope. Hierarchical file system. 11/16/2019. https://www.computerhope.com/jargon/h/hierfile.htm.
[29] Cmake. About CMake. https://cmake.org/overview/.
[30] Idil Fibres Optiques. Fiber Bragg Gratings (FBG) summary. https://www.idil-fibres-optiques.com/product/fiber-bragg-gratings/.
[31] MongoDN. JSON and BSON. https://www.mongodb.com/json-and-bson.
[32] Khushi Priya. Binary JSON (BSON). https://iq.opengenus.org/binary-json/.
[33] MongoDB. MongoDB CRUD Operations. https://www.mongodb.com/basics/crud.
[34] Shanika Wickramasinghe. MongoDB Compass: Using the Mongo GUI. October 14, 2020. https://www.bmc.com/blogs/mongodb-compass/.
[35] MongoDB. What is MongoDB Compass?. October 14, 2020. https://docs.mongodb.com/compass/current/.
[36] Microsoft. Microsoft HoloLens 2. https://www.microsoft.com/en-us/hololens