POLITECNICO DI TORINO

Master's Degree Course in Computer Engineering

Master's Degree Thesis

Innovative Visualization of Data Generated by Photonic Sensors

Supervisors
Prof. Bartolomeo Montrucchio
Dr. Antonio Costantino Marceddu
Dr. Alessandro Aimasso

Candidate
Sergio Schiavello

December 2022

Summary

Fiber Bragg Gratings (FBGs) sensors consist of a set of Bragg reflectors capable of reflecting a single wavelength of light, directly proportional to the dimensions of the grating, and transmitting all the others. Analyzing the offset of the reflected wavelength from the rest state, it is possible to infer a temperature variation, a strain applied to the sensor, or more. Being based on the use of photons, this type of sensor is advantageous in environments where the presence of electromagnetic interferences may cause measurement anomalies in conventional electronic systems. In addition, it is lightweight, can be multiplexed on a single transmission line, and is a passive component.

The applications for interfacing with FBG sensors often have a non-intuitive user interface and have substantial deficiencies in the functionality offered to the user. Due to the novelty of this type of sensor, which is still being researched, they lack the innovative capability of precisely correlating the measured values to physical events. Therefore, the main objective of this thesis is to create an innovative application, called FBG Data Analyzer, to display in real-time the data coming from a set of optical fiber sensors. This data is partially processed during acquisition and is shown in a comprehensive and human-readable manner. This thesis is part of a collaboration between the Department of Control and Computer Engineering (DAUIN) and the Department of Mechanical and Aerospace Engineering (DIMEAS) in the Inter-Departmental Center for Photonic Technologies context. DIMEAS mainly employs FBG sensors for structural integrity monitoring applied to aircraft, but it also uses this technology for measuring thermal balances of various components; these applications can be used for both prognostics and diagnostics purposes.

The project was preceded by a session of requirements collection from the stakeholder research team to highlight the needs and expectations concerning the final application. The requirements include viewing the wavelength peaks trend in the time domain and possibly performing a spectrum analysis of those time series to detect oscillatory components by applying the Fast Fourier Transformation (FFT) algorithm to the selected signals. Another essential requirement is that, given an empirically determined coefficient, it should be possible to correlate the variation of the wavelength peak value to the corresponding delta strain/temperature, where

the variation is the difference between the value reflected by the resting FBG sensor and the one currently measured. The property just described is the topic currently under study and validation by the involved research team. The final application, which is the objective of this thesis, may reveal itself insightful and help perform a more significant number of tests in a shorter time, thanks to the real-time visualization of data previously available only in post-analysis.

The desktop application has been developed and tested on Windows using the Qt open-source framework. One of the reasons for its choice is the simplicity of porting the application to different operating systems. Most of the User Interface (UI) space is occupied by charts built on top of the Qt Charts module; they have been heavily customized and improved to meet the operators' needs. Given the rich presence of configuration parameters, an automatic saving system was implemented. It makes use of a JSON file which is also editable with an external editor; this allows the operators to save different variations of the settings file in a repository to have ad hoc configurations for each ongoing measurement campaign.

Physical sensor data is captured by the SmartScan by SmartFibres interrogator, which is the unit responsible for transforming wavelengths into digital values. The data is then sent to a software called Middleware, which processes this data and sends it either to a MongoDB database or through a TCP socket. Thanks to MongoDB's Change Stream feature, the FBG Data Analyzer can obtain input data by actively waiting for changes to the current collection or by connecting to the TCP socket of the Middleware; the connection methods are mutually exclusive.

During the development phase, the functionalities of the software were tested using an interrogator emulator capable of generating random wavelength peaks for the configured sensors. In the final phase, two types of tests were carried out. The first involved measurements of temperature variations induced through a climatic chamber, while the second regarded measurements of the strain applied to a carbon fiber specimen, which can simulate the tail of Anubi, a model aircraft designed and assembled by the Icarus team of the Politecnico di Torino. During the tests, CPU and RAM usage was closely monitored and remained at constant levels.

Acknowledgements

First of all, I would like to thank prof. Bartolomeo Montrucchio for the opportunity of working on this project, and dr. Antonio Costantino Marceddu with dr. Alessandro Aimasso for their continuous support during the realization of this thesis.

A special thanks to my parents, Antonietta and Pino, and my brother Marco, who have always supported me throughout my life, sharing joys, sacrifices, and successes. It is thanks to them that I became the person I am today.

A big thanks goes to my girlfriend Lavinia, with whom I shared the last six years of my life and who has always been there for me during the highs and lows of this intense journey.

Finally, I would like to thank all my family members, friends, and colleagues who have supported me along the way.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

The thesis is contextualized in the broader interdisciplinary project PhotoNext. This project is a result of the collaboration between the Department of Control and Computer Engineering (DAUIN) and the department of Mechanical and Aerospace Engineering (DIMEAS), inside the Politecnico di Torino, as part of the Inter-Departmental Center for Photonic technologies. The research group was founded in 2017 to combine knowledge and skills in order to develop advanced optical components for telecommunication and industrial applications. In addition, the group could extend this expertise to other fields, like aerospace, sensing, structural monitoring, and more.

Figure 1.1: *PhotoNext logo* [1]

Part of the research effort is currently being employed to validate the sensing capabilities of a particular type of optical fiber sensor called Fiber Bragg Grating (FBG). This sensor can capture a change in temperature, strain, pressure, tilt, displacement, acceleration, load, and more; all that can be achieved by observing the refractive index variation inside the optical fiber core caused by such physical changes. Among the beneficial properties of the FBGs, like being lightweight, small in size, and having the capability of being multiplexed in a single transmission line, there is another one that is of greater interest for many fields: they are not

susceptible to electromagnetic interference; this is the significant advantage of being a photon-based technology.

The component responsible for generating the light to be sent through the fiber is called Interrogator. The unit injects an ultraviolet beam of light inside the optical fiber and then analyzes the wavelength that is reflected by the FBG. Then, it sends the collected information through an Ethernet cable to the Middleware, the software in charge of cleaning the data coming from the Interrogator and sending it to the architecture visualization layer; this piece of software is installed and run on a Raspberry Pi 3 Model B. The transmission to the visualization layer can be achieved by sending the sensor data directly to the visualization unit using a TCP socket or inserting it into a MongoDB database, from which the final application will retrieve it.

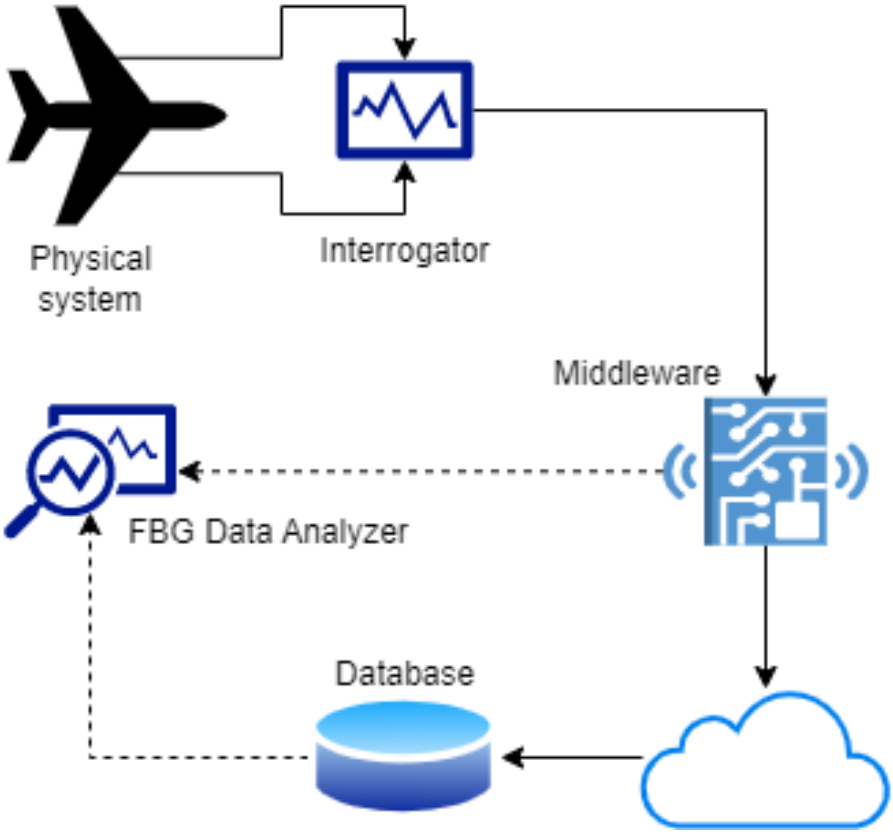

Figure 1.2: *System architecture*

At last, the data reach the visualization application used by the operator performing the measurement. Generally, the interrogator manufacturers provide the customers with software to visualize the data coming from their devices; however,

they understandably lack some cutting-edge features that may reveal very useful during the research work. One of them could be correlating the optical fiber refractive index variation to the physical variation in real-time, allowing the operators to fully leverage the sensing capabilities of the FBGs.

This thesis tries to fill the gap by creating a new GUI application with an improved user interface, encapsulating both legacy and innovative features. The application requirements have been discussed extensively with the researchers of the DIMEAS, the actual stakeholders in this project, and the final users of the software. They have also been integral to the development and testing phase, giving constant feedback to continuously validate the implemented features.

1.2 Fiber Bragg Grating

The FBG technology is based on a set of Bragg reflectors that, put together in a symmetric structure at a regular distance (pitch) one from the other, allow to reflect only the light at the Bragg wavelength while making all the others wavelengths pass through. It is possible to calculate the Bragg wavelength with the formula:

$$\lambda_B = 2n_{eff}\Lambda \tag{1.1}$$

where $2n_{eff}$ is the refractive index of the fiber core and Λ is the grating pitch.

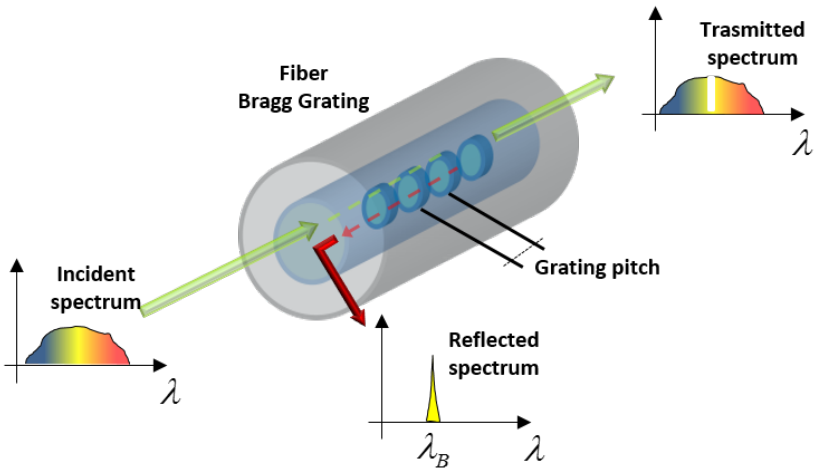

Figure 1.3: *FBG structure* [2]

As said before, different physical properties, like temperature or strain, can affect the refractive index of the silicate, which the reflectors are composed of, changing the Bragg wavelength reflected by the FBG.

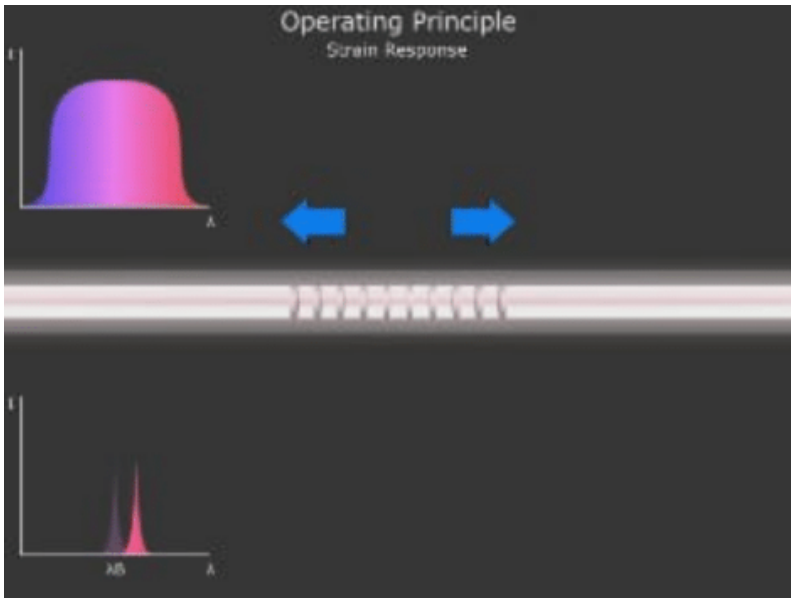

Figure 1.4: *FBG applied strain* [3]

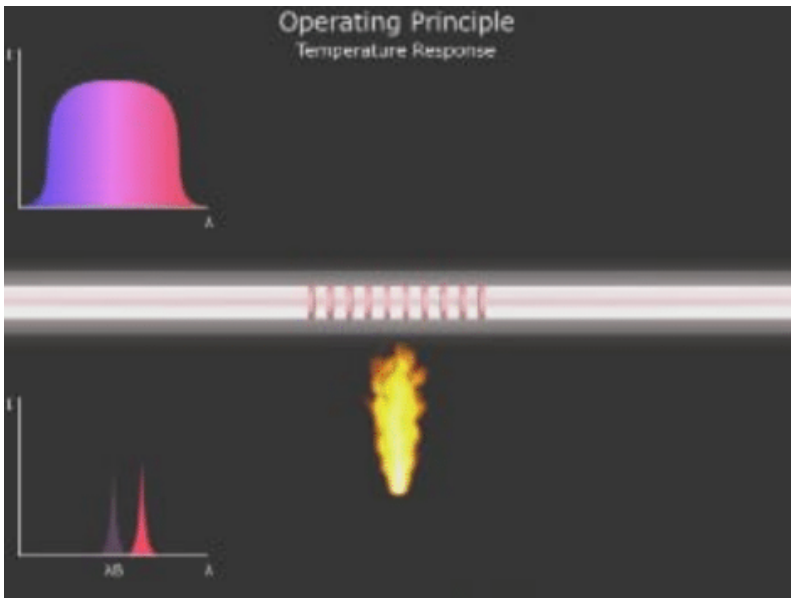

Figure 1.5: *FBG temperature change* [3]

When a change in the refractive index occurs, it can be calculated with the formula:

$$\frac{\Delta\lambda}{\lambda_0} = \frac{\Delta(2n_{eff}\Lambda)}{2n_{eff}\Lambda} \tag{1.2}$$

If the change actuator is a strain, the formula can be developed as follows:

$$\frac{\Delta\lambda}{\lambda_0} = \frac{\Delta(2n_{eff}\Lambda)}{2n_{eff}\Lambda} = (1 + p_e)\Delta\varepsilon = k\Delta\varepsilon \tag{1.3}$$

where p_e is the photoelastic constant (variation of the refraction index with axial tension), and k is the Bragg grating k factor.

In case of a temperature change, the formula takes the following form:

$$\frac{\Delta\lambda}{\lambda_0} = \frac{\Delta(2n_{eff}\Lambda)}{2n_{eff}\Lambda} = (\alpha + \xi)\Delta T \tag{1.4}$$

where α is the fiber thermal expansion coefficient, and ξ is the thermo-optic coefficient (dependence of the refraction index on temperature).

1.3 Thesis structure

The dissertation of this thesis is divided into the following chapters:

- Chapter 2: System architecture
- Chapter 3: User manual
- Chapter 4: Developer manual
- Chapter 5: Tests & results
- Chapter 6: Conclusions

Chapter 2

System architecture

The following chapter describes all the components involved in the workflow, from the physical system to be monitored to the visualization of the measured data in the terminal application.

2.1 System overview

The FBG Data Analyzer is implemented on top of the preexisting architecture designed and implemented by the PhotoNext group. Indeed, the data fed to the visualization layer is processed by different components in the underneath layers.

The system is composed of the units below, which work in synergy to complete the necessary tasks:

- Physical system;
- Interrogator;
- Middleware;
- Database;
- FBG Data Analyzer;

The coming sections will explain in detail what the different components in this pipeline do and how they cooperate to achieve the desired results.

2.2 Physical system

The physical system can contain the optical fibers hosting the FBGs; therefore, the entire architecture is system agnostic, and the FBGs can be considered the physical system. During the development and test phase, the systems monitored with the FBG Data Analyzer were an aircraft model (ANUBI) and a climatic chamber.

2.2.1 ANUBI

ANUBI is an aircraft model designed and assembled by the Innovation Center for Amateur Rocketry and Unmanned Ships (ICARUS) Team. The optical fibers are anchored to the wings and tail of the model using a specific glue; they are used to measure the strains applied to the aircraft to monitor its structural integrity and the physical forces it is subjected to.

Unfortunately, because of logistic issues, it was impossible to test the application in an actual flight, but the tests were performed with the real model in the laboratory.

Figure 2.1: *ANUBI aircraft model* [4]

2.2.2 Climatic chamber

A climatic chamber allows for varying and testing two quantities, temperature and humidity, in a confined space. The common tests performed with this instrument are the following:

- Thermostatic tests, wherein the only controlled parameter is the temperature;
- Climatic tests, wherein both the temperature and humidity parameters are controlled simultaneously;

For the purposes of this thesis, only thermostatic tests were conducted. The instrument has dedicated sensors to monitor the quantities mentioned above; therefore, it was used to validate the temperature readings of the FBG sensors.

Figure 2.2: *Climatic chamber* [5]

2.3 Interrogator

The Interrogator is the device in charge of emitting the laser beam into the fibers and reading back the reflected wavelengths of the FBG sensors.

Multiple products are available from different suppliers; the DIMEAS chose the SmartScan by SmartFibers interrogator, which is the product that was also used for the development and testing of the FBG Data Analyzer application.

The SmartScan has 4 channels, each hosting one optical fiber that can support up to 16 FBGs; therefore, it can read a maximum of 64 FBG sensors simultaneously connected. For each channel, the Interrogator can collect both raw and peak data, which are sent through the RJ45 Ethernet Connector embedded in the device using UDP packets.

Additionally, the device presents a serial port that can be used for diagnostics and servicing.

Figure 2.3: *SmartScan interrogator* [6]

2.4 Middleware

The Middleware is a multi-thread C++ command line application that receives the data generated by the Interrogator and elaborates it to make it usable by the visualization and database layer. This application generally runs on a Raspberry Pi 3 Model B connected with an Ethernet cable to the Interrogator.

At the beginning of the thesis work, the most updated version of the Middleware was only capable of sending the elaborated data to the database. In addition to its storage functionalities, it was also used as a "proxy" to exchange the FBGs data between the visualization layer and the Middleware.

During this thesis work, it was introduced a new connection method, already present in previous versions of the software. However, as will be discussed in the following chapters, the functionality was revisited and improved.

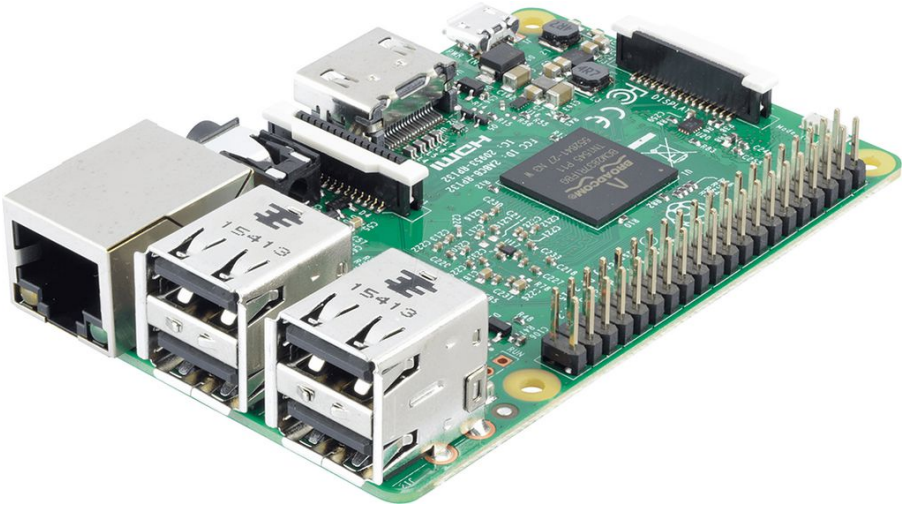

Figure 2.4: *Raspberry Pi 3 Model B* [7]

2.5 Database

Figure 2.5: *MongoDB logo* [8]

MongoDB is a non-relational, fast, and scalable database. These features, together with the so-called *Change Stream*, led it to be chosen for this project; this feature allows a client to listen for changes to a database or collection and be notified when they happen. This mechanism is used by the visualizer application to retrieve the FBG data from the database when this type of connection method is chosen.

As it is easy to imagine, the downside of this approach is the latency; indeed, it must be considered the time required to upload the data from the Middleware to the database, the time for the change stream notification to fire, and the time the data takes to be transferred from the database to the FBG Data Analyzer.

2.6 FBG Data Analyzer

The FBG Data Analyzer has been developed in C++ using the Qt framework. It displays the data read by the sensors in a very organized and human-readable manner, giving the operator insights into the acquired data in real-time.

The application retrieves the sensor data by connecting to the TCP socket server opened by the Middleware or by listening on the MongoDB Change Stream; this can be decided during the connection configuration phase. Independently from the connection method, the data received can also be saved locally, allowing the researchers to make custom analyses of the measured information.

The FBG Data Analyzer can display the instantaneous peak reflected by the sensors, the trend over time of the peaks, and its spectrum computed using the FFT algorithm. In addition, given the correct configuration, it is capable of calculating the change in temperature or strain correlated to the deltas of the peaks and displaying the current temperature or strain applied to the sensor with their trend over time.

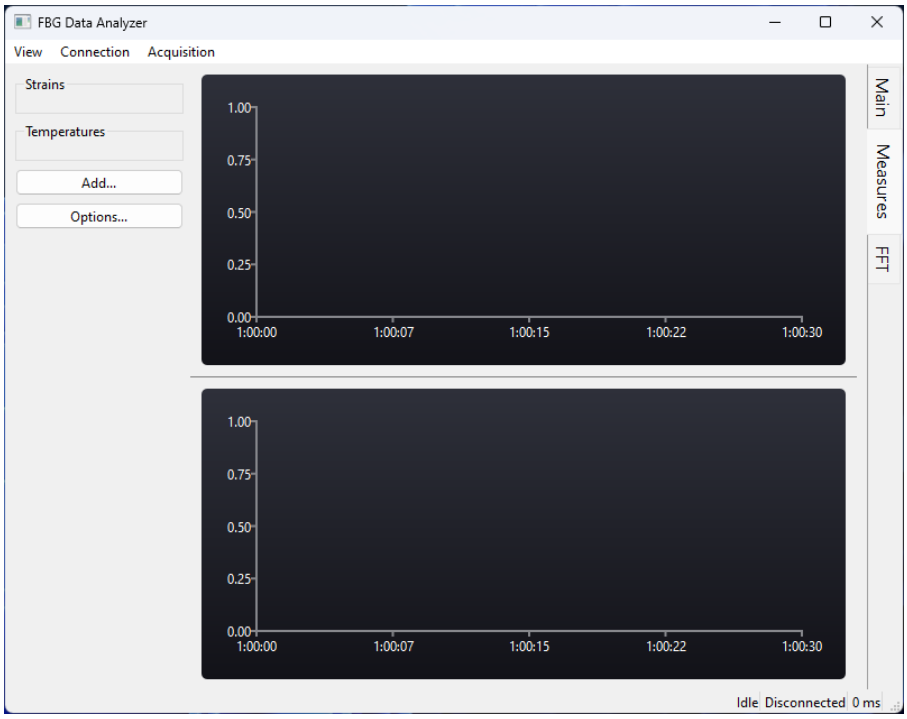

Figure 2.6: *FBG Data Analyzer*

Chapter 3

User manual

This chapter describes in detail the FBG Data Analyzer functionalities and guides the user in the configuration of the measurement session.

3.1 Overview

The application is divided into three different views, available in the lateral tab selector:

- **Main**: visualizes the instantaneous peaks and their trend over time;
- **Measures**: visualizes the temperature or strain over time measured by the linked sensors;
- **FFT**: visualizes the spectrum of the selected sensors;

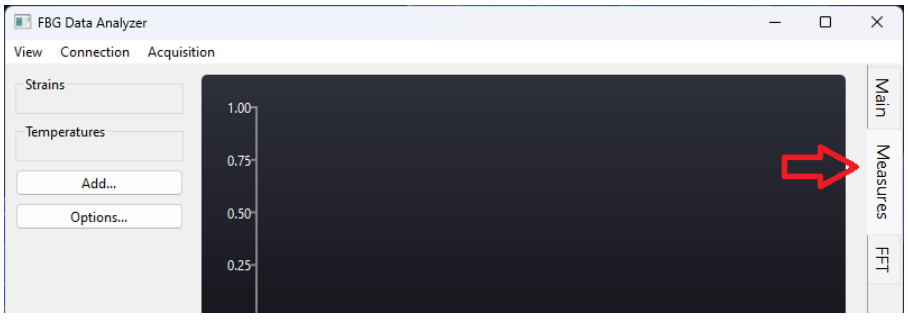

Figure 3.1: *Views tab*

Independently from the view selected in the main window, it is possible to open a separate window containing a table with the instantaneous peak of every sensor available.

3.2 Menu

The menu allows the user to open the table view window, manage the connection with the Middleware and configure the acquisition saving.

3.2.1 View

Clicking the *Table* subsection, it is possible to open and close the *Table view* window. The new window is detached from the main workspace; therefore, it is possible to move it around on the desktop, possibly also on another monitor, to always have an overview of the instantaneous values of the sensors, independently from the visualized data in the main window.

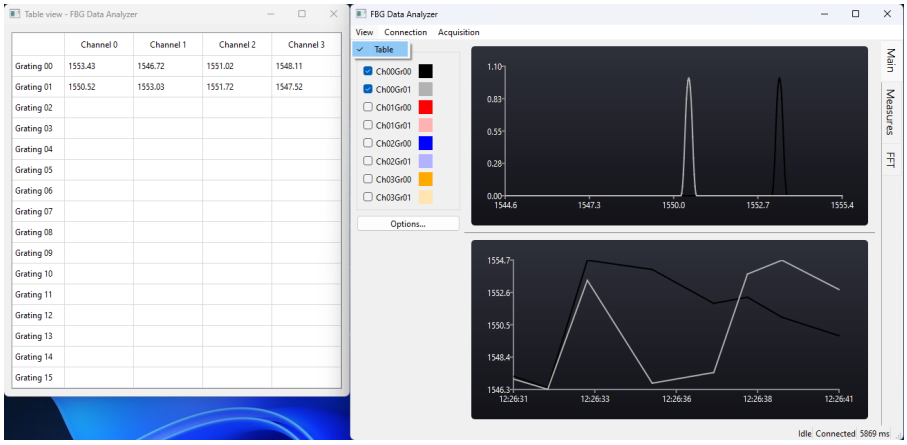

Figure 3.2: *Table view*

3.2.2 Connection

From this section, it is possible to select and configure the type of connection to be used for retrieving the sensor data from the Middleware; the options available are a TCP connection or a database proxy.

During the same session, it is possible to open and close multiple connections, even using different methods, without the need to close the application; only one open connection is allowed at any given time. Independently from the connection method, inside the status bar located in the bottom-right corner of the screen it is displayed the current connection status, which can be:

- **Disconnected**: the connection is closed, and no data is received;
- **Connecting**: the connection has not been established yet, but the client is actively trying to connect to the counterpart;

- **Connected**: the connection is open, and data is flowing into the application;

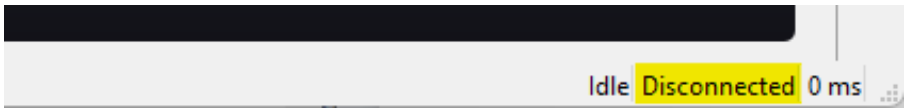

Figure 3.3: *Connection status*

TCP connection

The only required parameters for a TCP connection are the IP of the device on which the Middleware is running and the port to which the socket server has been bounded.

For obvious reasons, the machine running the FBG Data Analyzer and the device running the Middleware must be on the same network or be reachable via the Internet, even if the second option is not advised for latency reasons.

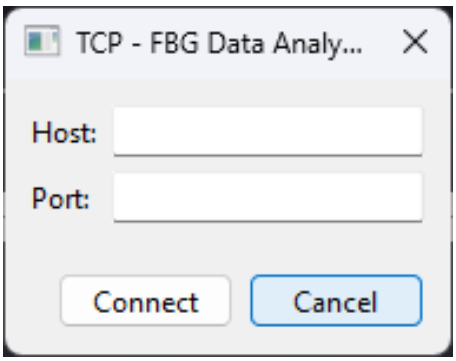

Figure 3.4: *TCP connection dialog*

Database connection

When using this type of connection, the data is not directly received from the Middleware, but it passes through a MongoDB instance that notifies and forwards the sensor data to the FBG Data Analyzer. This connection method can be advantageous when multiple users need to access the sensor data in a fast and easy way, as it is saved in a remote, quickly reachable, location.

Operators are allowed a high degree of flexibility in defining the parameters for connecting to the database. They are summarized below:

- **Host**: server IP or domain name;

- **Port**;
- **Database**: the database, inside the server instance, on which the Middleware writes the data received from the Interrogator;
- **Auth database**: the database that will be used to validate the credentials provided for the connection;
- **Username**;
- **Password**;
- **Direct connection**: allows the client to connect directly to the specified host as a standalone rather than to the replica set, which is the top-level abstraction of it;
- **SSL/TLS connection**: specifies if the connection should be established on top of an SSL/TLS secure channel;

As it is visible from Figure 3.5, the parameters *Auth database*, *Username*, and *Password* are optional and only required when an authenticated connection is needed.

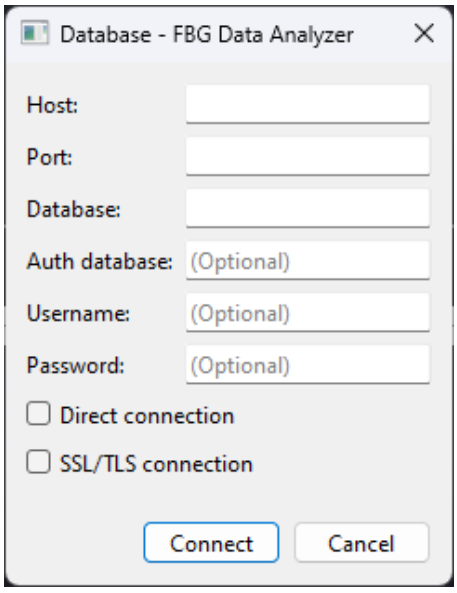

Figure 3.5: *Database connection dialog*

3.2.3 Acquisition

Within this section, it is possible to start the logging, stop it, and configure with which modality the sensor data received by the FBG Data Analyzer is saved on disk.

Opening the settings window, the operator can configure the following parameters:

- **Sampling**: at which rate the instantaneous peak will be sampled and saved;
- **Duration**: the duration of the acquisition. After that time, the logger is interrupted, and the log file is finalized;
- **Sensors**: the subset of the sensors to be logged;

Thanks to the easy-to-use input method, the sensors can be selected with the modalities below:

- **Individually**: clicking on the corresponding cell;
- **By channel**: clicking on the row label;
- **By grating index**: clicking on the column label;
- **By range**: clicking on a cell and dragging the cursor to another cell to form the wanted selection area;

By pressing the CTRL key, it is possible to combine multiple selection methods.

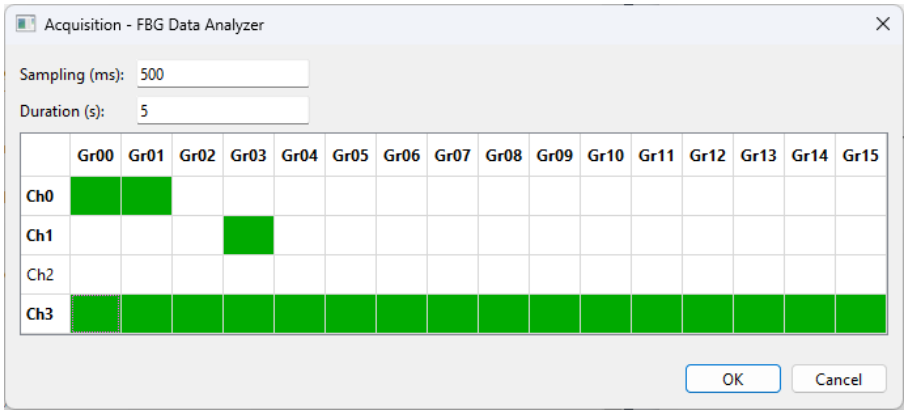

Figure 3.6: *Acquisition dialog*

After the configuration is confirmed, it is sufficient to click the *Start* button in the submenu to start the acquisition and the *Stop* button to stop it before the *Duration* period is elapsed; otherwise, the logger will automatically stop. The

logger status is easily obtainable with a glance at the status bar in the bottom-right corner of the application (Figure 3.7). The status can take the following values:

- **Idle**: the logger is not recording any data;
- **Recording**: the data received is currently being saved to disk;

The log files are saved in the folder *acquisitions* with an easily identifiable name composed of the date and time at which the acquisition was started (e.g., `20220918-165312.log`).

Figure 3.7: *Acquisition status*

3.3 Views

To easily cycle the different views, a tab system has been implemented on the right border of the application; this allows the user to quickly move from one set of information to the other. The only view not in the tab system is the *Table view*, which was considered more convenient to have as a separate window, as explained in Section 3.2.1.

3.3.1 Main view

The *Main view* plain (Figure 3.8) is divided into three sections. On the left is the sidebar containing all the available sensors, while on the top-right and bottom-right, there are the peaks and peak-time charts.

The vertical space occupied by the two charts can be customized by the user dragging up and down the horizontal divider in grey; the separator can be moved to the far-top or far-down positions to completely hide one of the two charts and focus the attention on a single one that will occupy the entire vertical space of the view (Figure 3.9).

Sidebar

In the sidebar is shown a legend with all the available sensors. This legend is automatically populated, as when the connection is established with the data source, a sensor discovery procedure takes place, adding each active sensor to the list. When

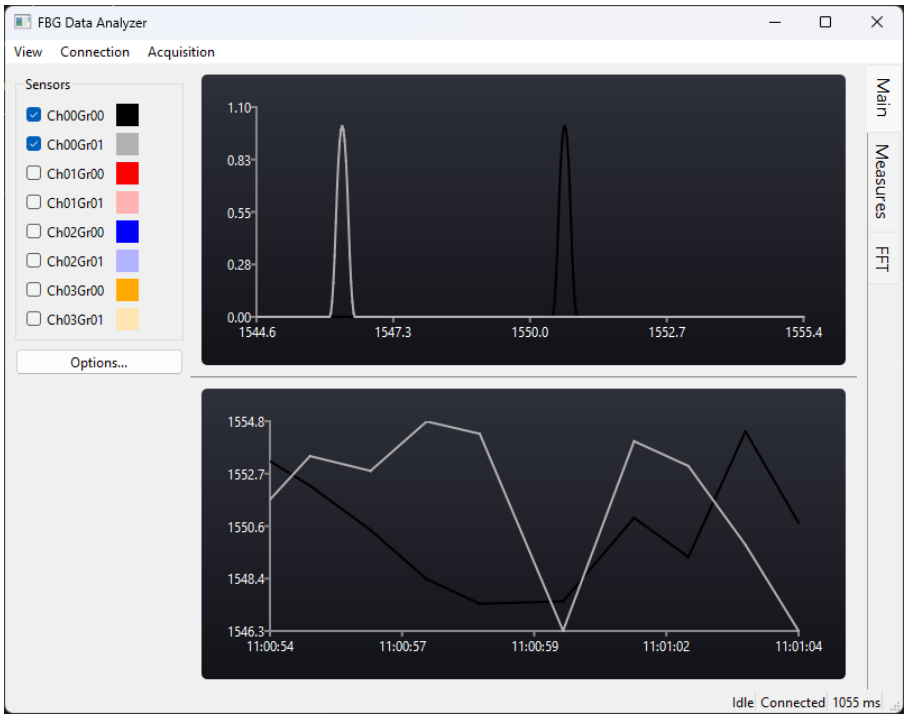

Figure 3.8: *Main view*

a sensor is added, it is shown by default in the charts; it can be easily hidden by unchecking the grating in the legend.

The sensors, color-wise, are grouped by channel; the system assigns a base color to it, and the gratings within are assigned a gradient level. The level is determined automatically based on the number of gratings present in the channel; for each sensor added, all the gradient levels assigned with the same base color are recalculated to always have an optimal level distribution.

Clicking on the color next to the sensor, a color picker will pop up, allowing the user to change the base color assigned to that channel. When the selection is confirmed, the colors of all the sensors inside the channel will be recomputed. The sensor colors are synced and reflected in every view.

Clicking the button below the legend makes it possible to open the charts options window (Figure 3.10), which allows tuning the charts setup based on the operator's specific needs.

While the majority of the options inside the window are specific for one of the two charts and will be covered in the following sections, the *Show chart grid* influences both the charts in the view, enabling or disabling the grid overlay that helps to get a quick qualitative estimate of the values in the charts (Figure 3.11).

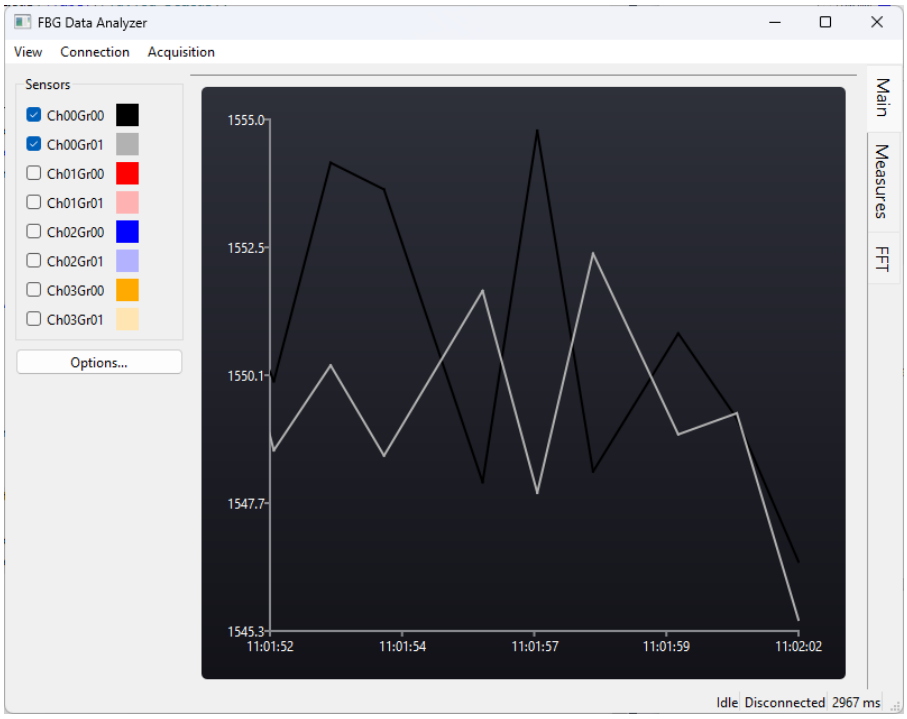

Figure 3.9: *Main view focused on one chart*

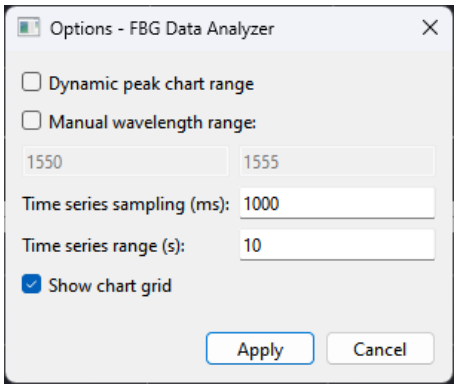

Figure 3.10: *Main view options*

Peaks chart

This chart visualizes the instantaneous peak of each selected sensor. The values on the y-axis are predefined because the intensity is not a piece of information available from the Middleware. At the same time, on the x-axis, the wavelength of the peaks is represented in nanometers (nm).

By default, the x-axis range is readjusted every time a peak is received, granting

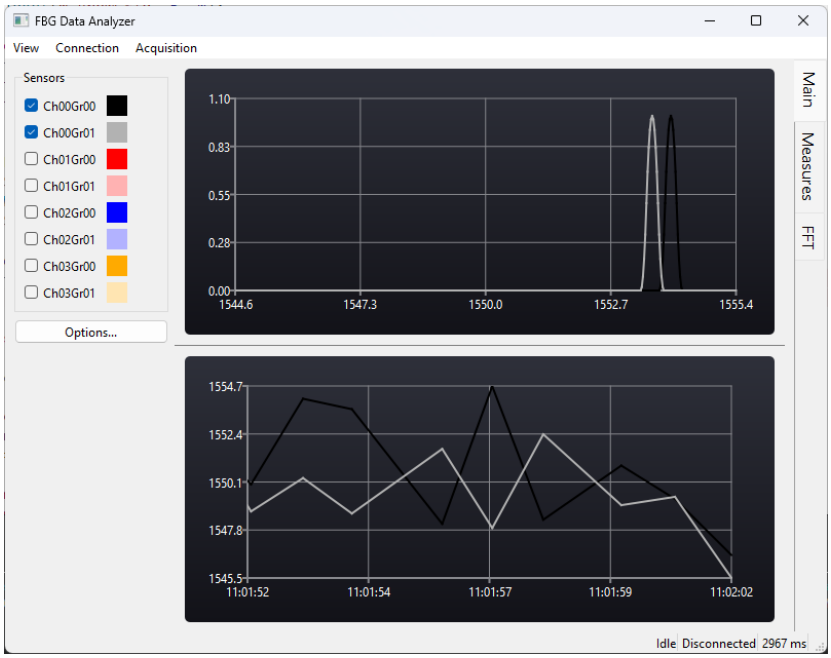

Figure 3.11: *Main view with chart grid enabled*

that every piece of information available is always visible on the charted area. The downside to this approach could be noticed in scenarios where significant sporadical variations can occur. Indeed, in that case, the useful segment of the range could become very small compared to the fully displayed range since the readjustment in this modality only increases it without ever shrinking it. For the described scenarios it was implemented the dynamic readjustment mode activated by the option *Dynamic peak chart range* (Figure 3.10). In this mode, the range is shrunk and expanded to always show all the available data without wasting horizontal space, since the current minimum and maximum peaks are always pinned to the start and the end of the x-range, respectively.

In addition to the table view, it is possible to get the precise peak wavelength value also from the chart; indeed, hovering with the mouse cursor on one of the series in the peaks chart, the series will be highlighted, and a label indicating the precise wavelength value will pop up near the peak of the series (Figure 3.12).

Peak-time chart

This chart visualizes the peak trend over time of the selected sensors. On the y-axis is reported the peak wavelength in nanometers (nm), while on the x-axis is shown the relative time in hours, minutes, and seconds. The y-axis range is automatically scaled to contain all the peaks in the charted timespan, but it can also be manually

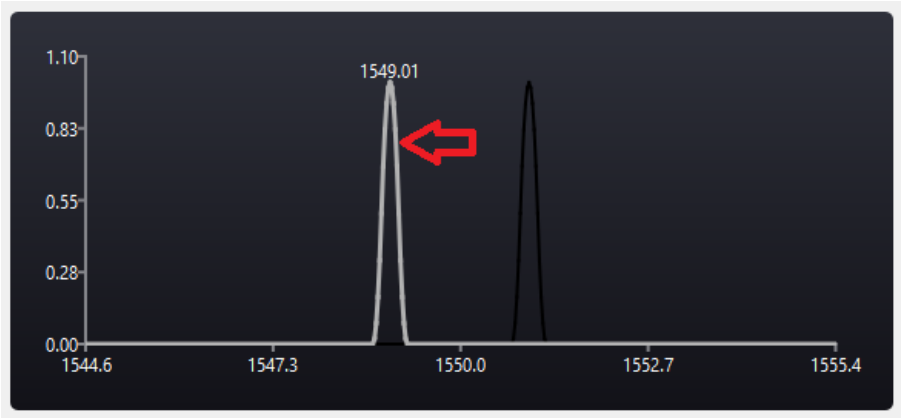

Figure 3.12: *Peaks chart with grey series highlighted*

configured with the *Manual wavelength range* option. The sampling frequency and the timespan visualized can be both changed with the options *Time series sampling* and *Time series range*, respectively (Figure 3.10).

It is possible to view older points in the series that are not included in the current window by hovering the chart with the mouse pointer and using the scroll wheel of the mouse or the touchpad scroll gesture. While scrolling, the y-range will dynamically adjust to include the minimum and maximum peak values in the x-axis range. To exit the scroll mode and go back to the live one, where the chart automatically scrolls to display the current instant, it is sufficient to scroll back to the last data points or to simply double-click on the chart.

Since the chart grid can only give a qualitative estimate of the coordinates of a point, a crosshair feature was embedded in every time chart. Keeping the mouse left button pressed inside the chart area, a crosshair will appear under the cursor position, projecting its exact coordinates to the axes (Figure 3.13); holding the button pressed and moving the mouse cursor around, it is possible to see the precise values of different data points in the past.

3.3.2 Measures view

Within the *Measures view* (Figure 3.14) are visualized the trend over time of temperatures and strains correlated to the variations of the sensor peaks. The area is divided into three sections: on the left is the sidebar, from which new sensors can be configured and added to the legend; on the top-right, it is possible to see the strains chart, while on the bottom-right, that of the temperatures. The vertical space of the charts can be customized as explained in Section 3.3.1.

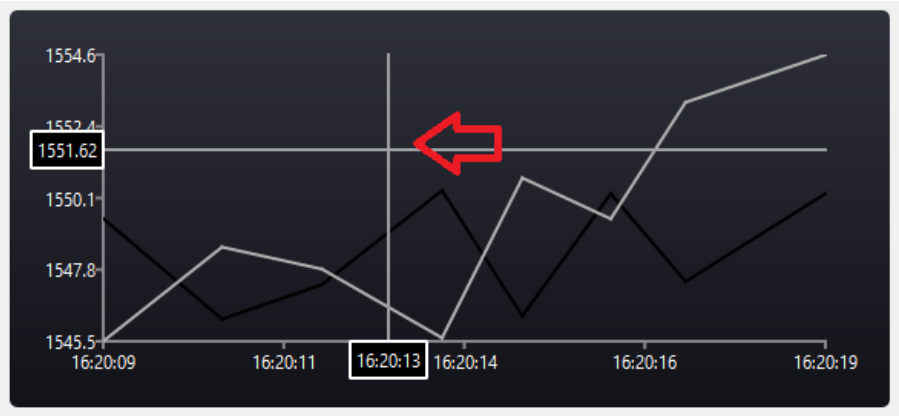

Figure 3.13: *Time chart crosshair*

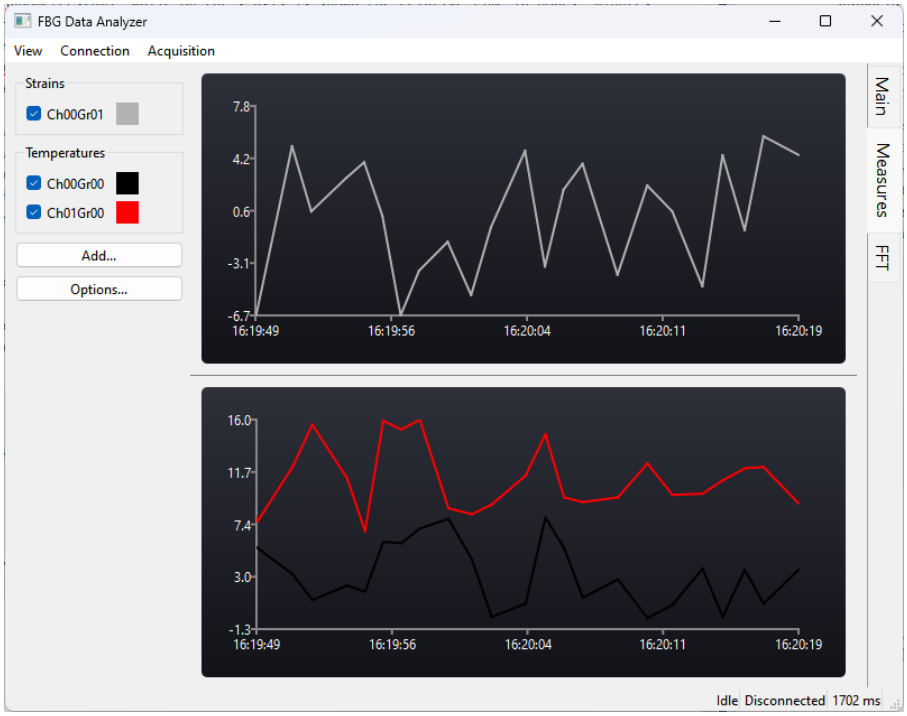

Figure 3.14: *Measures view*

Sidebar

The legend present in this view has the same feature as the one in the *Main view*, covered in Section 3.3.1, with a couple of differences. Initially, the legend starts without any sensor; the user must manually configure and add them to the view. For that reason, it is also possible to delete the sensors, for example, to correct

a mistake made during the configuration; to do that, it is sufficient to hover the sensor to be deleted with the mouse and click on the red *X* that will pop up next to it, as shown in Figure 3.15.

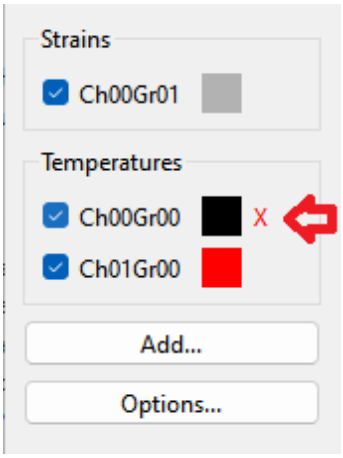

Figure 3.15: *Sensor deletion*

By clicking the *Add...* button, a new sensor can be configured to register a strain or a temperature. When configuring a new temperature sensor (Figure 3.16), two parameters must be specified:

- k_T: the experimentally determined coefficient correlating the $\Delta\lambda$ to the ΔT;
- T_0: the current temperature, used to calibrate the sensor;

When configuring a new strain sensor (Figure 3.17), the parameters to be specified may vary from two to three:

- k_ε: the experimentally determined coefficient correlating the $\Delta\lambda$ to the $\Delta\varepsilon$;
- ε_0: the current strain, used to calibrate the sensor;
- T Sensor: this field is optional, and if present, it indicates the grating to use as a temperature reference to compensate the strain read, since both temperature and strain variations have a role in the change of the Bragg wavelength.

Time charts

The temperature and strain charts have the same features and characteristics as the peak-time chart in the *Main view* and are described in Section 3.3.1. The only difference is the y-axis unit of measure, which may vary based on the one used for the calibration since the actual computed values are only deltas.

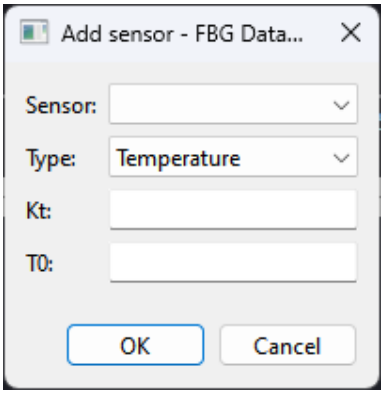

Figure 3.16: *New temperature sensor*

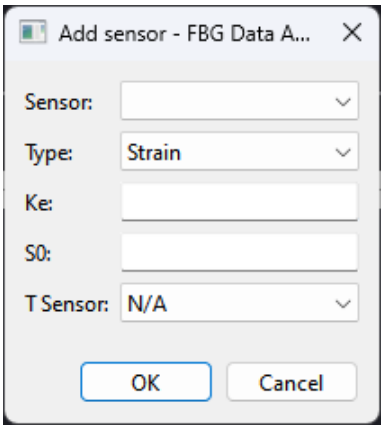

Figure 3.17: *New strain sensor*

3.3.3 FFT view

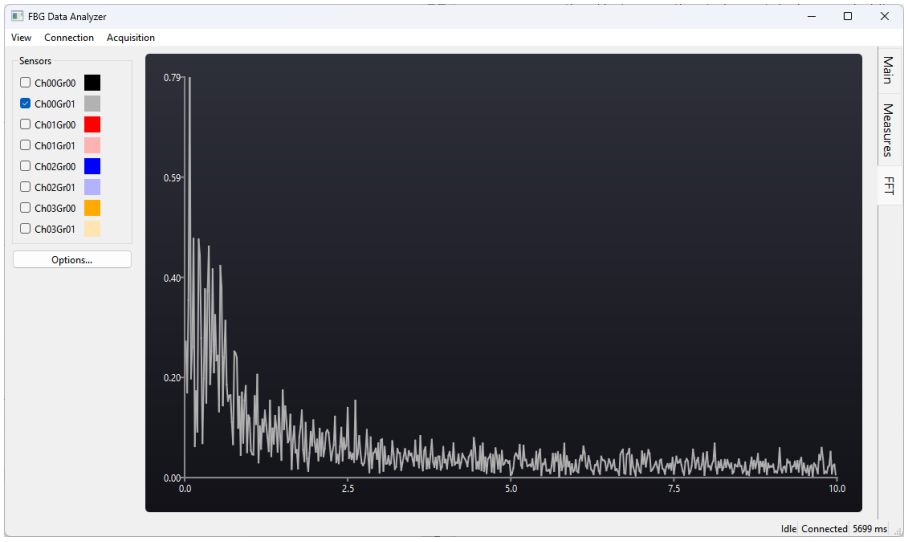

Figure 3.18: *FFT view*

This view (Figure 3.18) visualizes the spectrum of the selected sensors. It comprises two sections, the sidebar and a single chart occupying most of the view.

The spectrum is computed using the Fast Fourier Transformation (FFT) algorithm applied to a window of n samples; the window does not advance by one element at a time but in steps of n elements. Because of that, the FFT of the signal is not calculated every time a new sample is available but only when n new elements are ready to be utilized. This behavior was decided with the input of the research team involved in the requirements analysis and testing, and highly improves the computation performance.

The sidebar contains the legend, which works in the same way as the one in the *Main view*, analyzed in Section 3.3.1. For performance reasons, the spectrum analysis is only performed for the displayed sensors; when a sensor is hidden, its spectrum is kept in memory, but it will be recalculated only after the configured amount of samples is acquired from the time it is shown again.

Options

Thanks to the *Options* window, it is possible to configure the following parameters:

- **Plot direct component**: by default, the first value of the FFT result vector, which represents the direct component of the signal analyzed, is not plotted to make the chart more readable;
- **Sampling**: the frequency at which the analyzed signal will be sampled;
- **Samples**: the number of samples on which to compute the FFT (must be a power of 2);
- **Show chart grid**: displays a grid in the chart background.

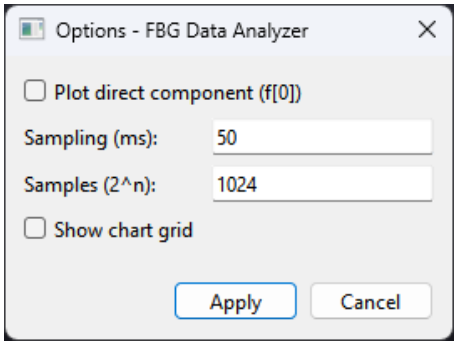

Figure 3.19: *FFT options*

3.4 Configuration

All the options and settings described in the sections above are automatically saved in the JSON file `settings.json` located in the FBG Data Analyzer installation folder. This is very convenient and allows the operators to change the configuration with an external editor without having to use the application GUI. Additionally, this saving method makes it very easy to back up the settings or create predefined templates to be used in similar measurement campaigns.

Chapter 4

Developer manual

This chapter describes the development setup and environment; the reader will be walked through the codebase architecture. This chapter can be a handy starting point for future developers who will continue working on this project.

4.1 Qt framework

Figure 4.1: *Qt logo* [9]

Qt is an open-source multi-platform framework that supplies developers with a toolkit to create applications for different platforms with a relatively low effort; indeed, a lot of Qt classes exist to cover multiple technological areas, from the Graphical User Interface (GUI) to the network communication, and wrap low-level platform-specific functionalities and APIs, offering a common and unique interface.

On top of that, different classes and external modules exist to facilitate specific aspects and needs that a developer might have; for instance, the charting system of the application is built on top of the Qt Charts module, while the socket communication is handled with the Qt Network one.

Specifically for the development of the FBG Data Analyzer, Qt v6.2.4 was used.

4.1.1 Signals & slots

In this framework, events are handled slightly differently from the approach of other ones; instead of using callbacks, as often done, Qt uses a signal/slot mechanism. In this methodology, an object that wants to let the outer world know that something occurred can register a signal that will be emitted when the event occurs; the signal is emitted a priori, as the object does not know about others that may be interested in the event. On the other hand, objects can also register slots, which are practically public member functions that have the ability to be attached and react to specific signals. Finally, signals and slots can be connected together with the function `connect()` so that a slot is triggered when a signal is emitted; the connected signal and slot do not have to be owned by the same object. Multiple slots can be connected to the same signal, and symmetrically, a single slot can be connected to various signals.

As will be presented in the following sections, the FBG Data Analyzer heavily uses this system to handle events.

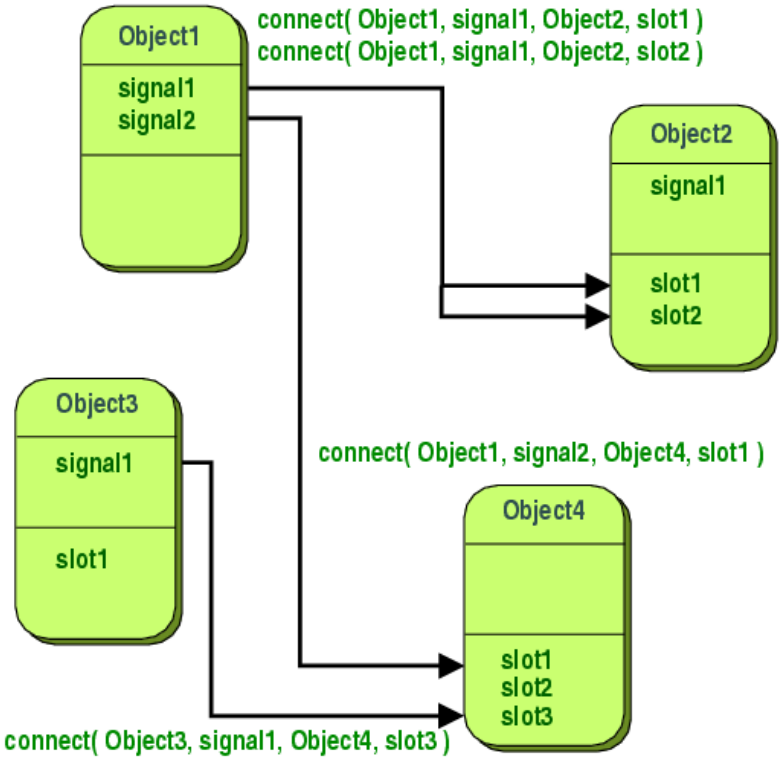

Figure 4.2: *Signals and slots connection* [10]

4.1.2 GUI definition

The three key classes that define a GUI in Qt are the following:

- `QWidget`: the base class of every drawable class that represents a base element of the GUI. Every `QWidget` can be displayed as a window if it does not have a parent; otherwise, it will be rendered inside the parent `QWidget`;
- `QMainWindow`: a special class that describes the main window of the application that is opened at start-up. For this reason, it is designed to allow an easy presentation of a menu bar, a central widget, and a status bar;
- `QLayout`: a class that every `QWidget` containing multiple elements can use to organize how those subcomponents should be displayed in the view.

To control the appearance of every component, the framework introduced the Qt-StyleSheet (QSS) language, inspired by the Cascading Style Sheets (CSS) language, which is commonly used for defining the style of HyperText Markup Language (HTML) pages; indeed, QSS is straightforward to learn, having prior knowledge of CSS.

4.1.3 Memory management

The `QObject` is the class from which every class inside the Qt framework inherits. Instead of instantiating the objects in the stack, Qt heavily uses dynamic memory allocation; for this reason, at first glance, it may seem an arduous job to keep everything that must be handled and deleted under control, making the system look prone to memory leaks and more. That is not the case as `QObjects` are organized in hierarchies, where every object that is instantiated is added to the children's list of the parent; when the parent is deleted, every child in the tree will automatically be deleted.

Therefore, it is common to find in the codebase function calls that have as arguments dynamic instantiations of objects without having the allocated memory registered in the caller object. Ownership of the new entity will directly pass to the callee, that will take care of deleting it when suitable.

4.2 MainWindow

The `MainWindow` is the class that describes the main window of the application that is opened at the program launch by the entry point `main` function (Figure 4.3) and where all the user-defined components reside.

All the views are initialized inside the constructor of this class (Figure 4.4), along with the global variables that need to be accessed by most of the classes.

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

Figure 4.3: *Main function*

```
ConnectionManager g_connection;
CustomStatusBar *g_statusBar;
SessionSettings *g_settings;

MainWindow::MainWindow(QWidget *parent)
    ...
{
    ...
    g_settings = new SessionSettings();

    QTabWidget *tabView = new QTabWidget(this);
    tabView->addTab(new PeakTimeView(tabView), "Main");
    tabView->addTab(new MeasuresView(tabView), "Measures");
    tabView->addTab(new FFTView(tabView), "FFT");
    ...
    g_statusBar = new CustomStatusBar();

    setCentralWidget(tabView);
    setMenuBar(new CustomMenuBar());
    setStatusBar(g_statusBar);
    setWindowTitle("FBG Data Analyzer");
}
```

Figure 4.4: *MainWindow constructor*

The views are those described in the preceding chapter, while the global variables are instances of the following classes:

- `ConnectionManager`, which manages the connections to the database or the TCP server;

- `CustomStatusBar`, which represents the status bar at the bottom of the application;
- `SessionSettings`, which handles the automatic save and load of the settings available;

4.3 SessionSettings

Every option in the FBG Data Analyzer is automatically saved upon submission by the user; this also includes the selected sensors in the various chart legends, along with every channel color, independently from the selection state.

All this is handled with the SessionSettings class, which keeps track of the options in a JSON document stored in the private variable `m_settings`. This variable is initialized in the constructor by calling the function `loadSettings()`, which reads the content of the `settings.json` file and parses it in a JSON document. The class exposes some getters and setters that allow other objects to access specific document sections; an example is presented in Figure 4.5. When the `SessionSettings` object is destroyed, the `settings.json` file is overwritten with the content of the `m_settings` variable using the `writeSettings()` function.

Since the channel colors are synced between all the chart legends in the application, this class is also responsible for notifying all of them when a color is updated; this is achieved by emitting the signal `sensorColorChanged()`. When the `saveSensorColor()` (Figure 4.7) function is called, the new base color selected for the channel is passed as an argument; it then recalculates the individual sensor colors and emits the signal for each one of them. The gradient steps are determined by evenly distributing the useful gradient range among the sensors contained in the `m_gratings` map, which contains the number of gratings present in every channel. That map is kept updated by the `getSensorColor()` function (Figure 4.6) that is called by the chart legends every time a new sensor is detected.

4.4 ConnectionManager

The `ConnectionManager` acts as a proxy and point of access for every type of connection method currently used. It is an abstraction designed to facilitate the reception of sensor data for the components that need it; in this way, the classes that consume the data can be connection-method agnostic and just have a single interface to handle the connections.

The class exposes slots and signals that are then internally routed to the current instance of the `ConnectionInterface` pointed by the private variable `m_conn`. The only slots that differ based on the connection method are the ones that open it with the counterpart, `mongoConnect()` and `tcpConnect()`. When one of these methods

```
void SessionSettings::saveDbInfo(QString host, int port,
   QString db)
{
    auto &connection = m_settings["connection"]["db"];
    connection["host"] = host.toStdString();
    connection["port"] = port;
    connection["database"] = db.toStdString();
}
...
QVector<QVariant> SessionSettings::getDbInfo()
{
    ...
    auto &connection = m_settings["connection"]["db"];
    auto host =
       QString(connection["host"].get<std::string>().c_str());
    auto port = connection["port"].get<int>();
    auto database =
       QString(connection["database"].get<std::string>().c_str());
    return { host, port, database };
    ...
}
```

Figure 4.5: *Settings getter & setter*

```
QColor SessionSettings::getSensorColor(int channel, int
   grating)
{
    if(!m_gratings.contains(channel))
        m_gratings[channel] = grating + 1;
    else if(grating >= m_gratings[channel])
        m_gratings[channel] = grating + 1;
    ...
}
```

Figure 4.6: *getSensorColor function*

is called, the current connection is closed, and the `m_conn` object is deleted; once that is done, a new connection is opened using the method implicitly chosen by calling the respective slot, and the new instance is saved again in the `m_conn` pointer. At this stage, all the signals and the `disconnect()` slot exposed by the class are

```
void SessionSettings::saveSensorColor(int channel, int
    grating, QColor color)
{
    float step = m_gratings[channel] > 1 ? GRADIENT_RANGE /
        (m_gratings[channel] - 1) : 0;
    for(int gr=0; gr<m_gratings[channel]; gr++)
    {
        float p = 1 - step * gr;
        int r = color.red() * p + 255 * (1 - p);
        int g = color.green() * p + 255 * (1 - p);
        int b = color.blue() * p + 255 * (1 - p);
        emit sensorColorChanged(channel, gr, QColor(r, g, b),
            color);
    }
}
```

Figure 4.7: *saveSensorColor function*

wired with `m_conn` again by calling the `makeConnections()` function (Figure 4.8), while the consumers continue to receive the signals containing the sensor data as anything happened because the object they are connected to never changed. Looking at the example in Figure 4.9, the links that are detached and reconnected when the connection method changes are the ones between the `ConnecetionManager` and the `ConnectionInterface` derivates on the right, while the connections on the left remain in place for the entire life cycle of the consumers, even during a connection change.

The actual classes that directly handle the connections are `MongoInterface` and `TCPInterface`; these two classes both inherit from the abstract class `ConnectionInterface` and implement its pure virtual slots, `connect()` and `disconnect()`.

4.4.1 MongoInterface

This class manages the physical connection with the MongoDB instance using the C++ driver officially provided, called `mongocxx`. The driver builds on top of the C driver `libmongoc` and uses the matching BSON package `bsoncxx`, which implements the BSON specification, used for handling the data exchanged with the database.

```
void ConnectionManager::makeConnections()
{
    connect(m_conn, &ConnectionInterface::statusChanged,
        this, &ConnectionManager::statusChanged);
    connect(m_conn, &ConnectionInterface::peakReceived, this,
        &ConnectionManager::peakReceived);
    connect(m_conn, &ConnectionInterface::connectionFailed,
        this, &ConnectionManager::connectionFailed);
    connect(m_conn, &ConnectionInterface::configReceived,
        this, &ConnectionManager::configReceived);
    connect(m_conn, &ConnectionInterface::latencyComputed,
        this, &ConnectionManager::latencyComputed);
    connect(m_conn, &ConnectionInterface::connectionClosed,
        this, &ConnectionManager::connectionClosed);
}
```

Figure 4.8: *makeConnections function*

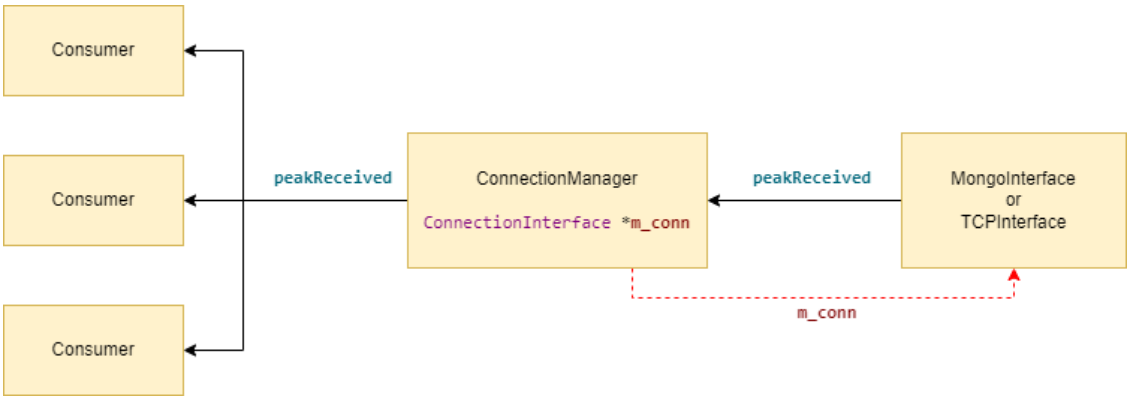

Figure 4.9: *ConnectionManager schema*

Status thread

When the slot `connect()` is called, the thread `m_statusTh` is started, and the `m_status` is changed to *Connecting*; this thread runs in a loop until the `disconnect()` slot is called. At every cycle, it tries to ping the database to check the connection status; if the server is unreachable, it enters/maintains the status *Connecting* and retries to ping the server after one second; otherwise, if the connection is established, it enters/maintains the *Connected* status and, if it was not already running, it starts the `m_streamTh` thread.

Stream thread

The `m_streamTh` runs in a loop as long as the interface status remains *Connected*. It reads the MongoDB *Change Stream* (Section 4.4.1) and elaborates the data received (Figure 4.10). If a new peak is obtained, it emits the signal `peakReceived()` to inform the interested consumers. If the elapsed time since the last latency computation exceeds the `LATENCY_FREQUENCY` definition, it also calculates the difference between the current timestamp and the one at which the Interrogator measured the peak, emitting the signal `latencyComputed()` with the result.

Instead, if a configuration message is received, it adds the new sensor to the detected gratings list and informs the system emitting the signal `configReceived()`.

The configuration message is only transmitted one time when the Middleware starts the measurement. This may cause a problem with the auto-detection mechanism in case the FBG Data Analyzer connects to the database after that record is generated. For this reason, when a new peak of a sensor that was not passively detected is received, the application calls the function `fetchConfig()` to execute a query on the database that fetches all the configuration entries available in the collection; the resulting gratings are inserted in the list of detected sensors and the `configReceived()` signal is synthetically emitted for each one of them.

MongoDB change streams

As cited above, the *Stream thread* uses the MongoDB Change Stream feature to get sensor data. Change streams allow applications to access real-time data changes without the complexity and risk of tailing the oplog. Applications can use change streams to subscribe to all data changes on a single collection, a database, or an entire deployment, and immediately react to them. Because change streams use the aggregation framework, applications can also filter for specific changes or transform the notifications at will [11].

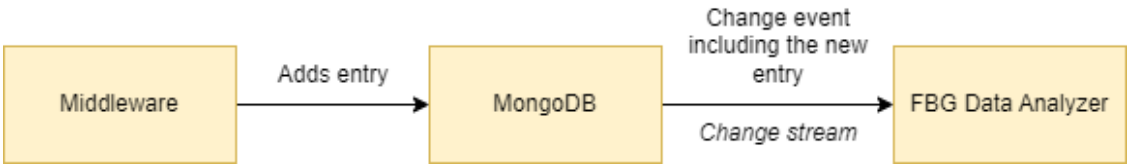

Figure 4.10: *Change Stream schema*

4.4.2 TCPInterface

This class handles the direct connection with the Middleware through a TCP socket; it uses the socket wrapper `QTcpSocket`, shipped with the Qt framework.

Socket connection

The socket instance is kept in the private pointer variable `m_socket`. When the `connect()` slot is called, the `m_status` is changed to *Connecting* and the `m_socket` slot `connectToHost()` is called to attempt to establish the connection with the server. When the socket connection state changes, the `m_socket` emits the signal `stateChanged()`, including the current state of the socket. If the socket is in the `UnconnectedState`, the `TCPInterface` object enters/maintains the status *Connecting* and the `connectToHost()` slot is called again, creating a non-blocking event loop entirely based on the slots and signals of the `QTcpSocket` class (Figure 4.11); the loop is only interrupted when the connection is established or the `disconnect()` slot of the `TCPInterface` is called. On the other hand, if the `stateChanged()` signal returns `ConnectedState`, the `m_status` is changed to *Connected*, and the `m_parserTh` thread is started.

```
QAbstractSocket::connect(m_socket, &QTcpSocket::stateChanged,
    [this](QTcpSocket::SocketState state){
     if(m_status == status::disconnected)
         return;

     switch(state)
     {
     case QTcpSocket::ConnectedState:
         changeStatus(status::connected);
         m_parserTh->start();
         break;
     case QTcpSocket::UnconnectedState:
         changeStatus(status::connecting);
         ...
         m_socket->connectToHost(m_host, m_port,
            QTcpSocket::ReadOnly, QTcpSocket::IPv4Protocol);
         break;
     }
});
```

Figure 4.11: *Handler of the stateChanged signal*

Once the socket connection is established and it is ready to read the data on the communication channel, the `m_socket` emits the signal `readyRead()` and continues to do so as long as data is received. When this is triggered, all the available data on the socket, which is JSON formatted, is read and put in the `m_queue` for further elaboration by the consumer `m_parserTh`.

Parser thread

This thread runs in a loop as long as the `m_status` is *Connected*. In every cycle, the thread extracts all the complete JSON objects available from the `m_queue` and elaborates them singularly.

Every time a new socket connection is established, the first message sent by the Middleware includes the active gratings detected; for every sensor in the array, the `configReceived()` signal is emitted to inform the system. The subsequent messages are sensor peak information, which is forwarded to the interested consumers with the signal `peakReceived()`. Analogously to the `MongoInterface`, when a new peak is received and the `LATENCY_FREQUENCY` time is elapsed from the last time the latency was computed, a new latency calculation is done and emitted with the `latencyComputed()` signal.

4.5 ChartLegend

This class implements the legend widget in the sidebar of every view, which shows the available sensors, allows to hide or show them, and changes the channel colors. This object is also the only one that interacts directly with the series inside the charts. Its constructor takes in input the following parameters:

- `SessionSettings::viewEnum view`: the type of view in which this legend is placed; this is used to save the configuration in the correct section of the `settings.json` file;
- `QString name`: the name that will be shown above the legend in the GUI;
- `QList<BaseChart*> charts`: the list of actual charts that the legend needs to control;
- `bool editable`: `false` by default, it indicates if the list of sensors can be changed by deleting them; for example, this is set to `true` in the *Measures view*.

The `ChartLegend` class implements only the slot `addSensor()` that takes in input the channel and grating of the sensor to be added to the legend; this slot is generally connected to the `configReceived()` emitted by the `ConnectionManager` and is the one that tells the charts to create a new series for the sensor (Figure 4.12). When the slot is called, a new `LegendRow` is created, which includes the `QCheckBox` to show and hide the sensor, the sensor name, and the `ColorPicker` to change its channel color; the checkbox and the color picker are initialized based on the values that are contained in the `g_settings` variable, fetched with the functions `getSensorVisibility()` and `getSensorColor()` respectively.

```
PeakTimeView::PeakTimeView(QWidget *parent)
{
    ...
    PeakChart *chart1 = new PeakChart();
    TimeChart *chart2 = new TimeChart();
    ...
    // Sidebar
    ChartLegend *legend = new
       ChartLegend(SessionSettings::PeakTimeView, "Sensors",
       {chart1, chart2});
    connect(&g_connection,
       &ConnectionManager::configReceived, legend,
       &ChartLegend::addSensor);
    ...
}
```

Figure 4.12: *ConnectionManager & ChartLegend connection*

The `colorChanged()` signal of the `ColorPicker` object is connected with a handler that calls the `saveSensorColor()` slot of `g_settings`, which internally emits the `sensorColorChanged()` signal to inform all the other legends and charts that the sensor colors were changed (Figure 4.7). To implement this mechanism, the `ChartLegend` object is also connected to the last mentioned signal; the event loop that those interconnections would cause is blocked thanks to the check on the `master` color that is passed as an argument to the `ColorPicker::setColor()` function (Figure 4.13).

```
void ColorPicker::setColor(const QColor &color, const QColor
   &master, bool set)
{
    if(master != m_color)
    {
        setPalette(QPalette(color));
        if(set) m_color = master;
        emit colorChanged(master);
    }
    else setPalette(QPalette(color));
}
```

Figure 4.13: *setColor function*

4.6 BaseChart

All the charts visible in the views are instances of classes derived from the `BaseChart` (Figure 4.14), which itself extends the `QChartView` implemented in the Qt Charts module. This class is the owner of the sensor series, which are saved in the map `m_series`, and exposes the slots showed in Figure 4.15. Additionally, it defines the protected virtual methods `onShowSensor()` and `onRemoveSensor()` that the derived classes can implement to perform some actions when a sensor is shown/hidden or it is permanently removed from the chart. The `BaseChart` class does not actually add or remove any data points to the series; this task is delegated to the subclasses.

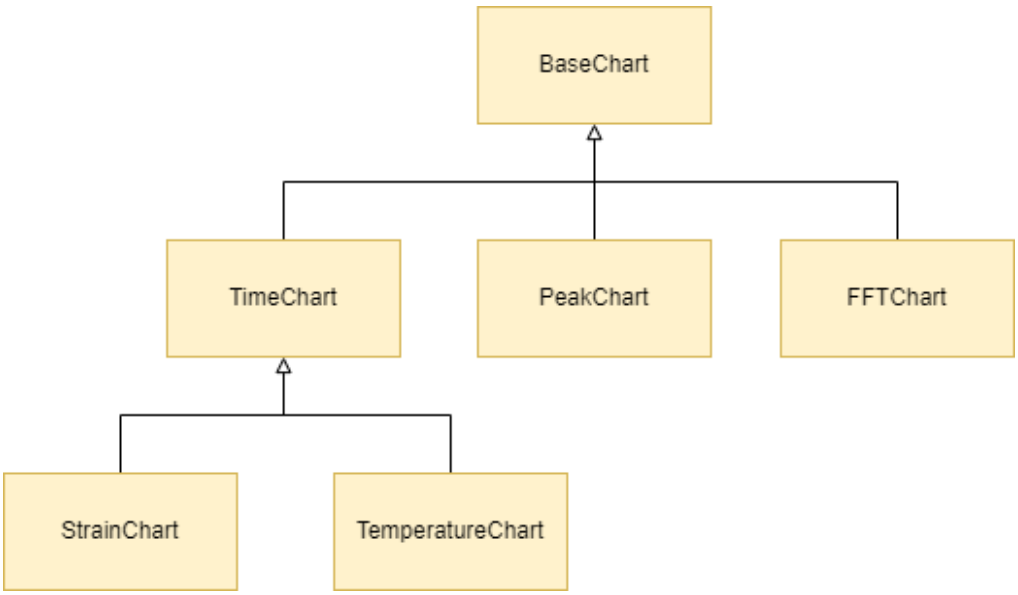

Figure 4.14: *BaseChart derived classes*

4.6.1 PeakChart

The peak chart is used to display the instantaneous sensor peaks that are received by the application. In the constructor, the `peakReceived()` signal of the `g_connection` is connected to the `newPeak()` class slot. When a new peak is received, a series is constructed by setting the extremes to zero and creating a peak with the shape of the `QEasingCurve::InOutQuad`, with the highest point coinciding with the sensor peak (Figure 4.16). At the same time, if the peak received is the first one of the sensor, a label with the peak value is created and placed above the peak point; the label is hidden by default and is only shown when the series is hovered with the mouse. In case the sensor already has a label in the chart, its text

```
class BaseChart : public QChartView
{
...
protected:
    QMap<QPair<int, int>, QLineSeries*> m_series;
    virtual void onShowSensor(int channel, int grating, bool
       show) {};
    virtual void onRemoveSensor(int channel, int grating) {};

public slots:
    void addSensor(int channel, int grating, const QColor
       &color);
    void showSensor(int channel, int grating, bool show);
    void setColor(int channel, int grating, const QColor
       &color);
    void removeSensor(int channel, int grating);
    void showChartGrid(bool state);
};
```

Figure 4.15: *BaseChart definition*

is updated along with its position in the chart to keep it aligned with the correct wavelength value. The labels are created and updated by calling the `setLabel()` method.

Range adjustment

To keep all the peaks inside the charted area, the x-axis range is constantly adjusted by calling the `adjustRange()` method (Figure 4.17). When a new data point is added to the chart, the method checks if the new value is contained in the current range by confronting it with the `m_min` and `m_max` variables, that store the extremes of the range and the sensors that defined them; if this is not the case, one of the two extremes of the range is expanded to include the new point, and the variables `m_min` and `m_max` are accordingly updated.

It is easy to see how an outlier could skew the range permanently, reducing the legibility of the chart. To address this issue, the *dynamic range* option was introduced; the option is active when the `m_dynamicRange` variable is set to `true`. During the time the option remains active, in addition to the standard extremes check, it is also checked if the sensor that generated the new peak is the one that determined one of the extremes; if the condition is met and the new value is inside the current range, the sensor with the new highest or lowest value is searched in the

```
void PeakChart::newPeak(int channel, int grating, qreal peak)
{
    ...
    if(series->isVisible())
        adjustRange(sensor, peak);

    QEasingCurve curve(QEasingCurve::InOutQuad);
    series->append(1000, 0);
    for(int i=10; i>0; i-=1){
        float fI = i / 10.0;
        float val = curve.valueForProgress(1 - fI);
        series->append(peak - fI/4, val);
    }
    for(int i=0; i<=10; i+=1){
        float fI = i / 10.0;
        float val = curve.valueForProgress(1 - fI);
        series->append(peak + fI/4, val);
    }
    series->append(2000, 0);
    setLabel(channel, grating);
}
```

Figure 4.16: *PeakChart::newPeak*

`m_peaks` map, which contains the latest peak of each sensor. The found sensor/peak pair is set as the new `m_min` or `m_max`, suitably updating the range. This method will always keep the sensors with the minimum and maximum peaks anchored to the extremes of the x-axis range.

4.6.2 TimeChart

The time chart is used to visualize all the trends over time. New peaks are received from the `g_connection` object by connecting its `peakReceived()` signal to the `newPeak()` slot. When a new peak is received, its timestamp is confronted with the one inside the `m_lastSensorUpdate` map; if the elapsed time is longer or equal to the `m_sampling` time, it is passed to the `toChartValue()` virtual method that converts the peak into the actual value to be added to the `m_series` and the value in the `m_lastSensorUpdate` is refreshed. The cited virtual method is intended to be overridden by the derivate classes; by default, it returns the value passed as an argument.

```
void PeakChart::adjustRange(QPair<int, int> sensor, qreal
    peak)
{ ...
    if(peak < m_min.second || m_min.second == 0)
    {
        xAxis->setMin(peak - PEAK_OFFSET);
        m_min = {sensor, peak};
    }
    if(peak > m_max.second)
    {
        xAxis->setMax(peak + PEAK_OFFSET);
        m_max = {sensor, peak};
    }

    if(m_dynamicRange)
    {
        if(sensor == m_min.first && peak > m_min.second)
        {
            m_min = min();
            xAxis->setMin(m_min.second - PEAK_OFFSET);
        }
        if(sensor == m_max.first && peak < m_max.second)
        {
            m_max = max();
            xAxis->setMax(m_max.second + PEAK_OFFSET);
        }
    }
}
```

Figure 4.17: *PeakChart::adjustRange*

Range adjustment

The `adjustRange()` method (Figure 4.18) is called every time a new peak is registered. As done in the `PeakChart` class, it resizes the vertical range to include every visible point. A peculiarity of this chart is that the user can manually fix the y-axis range; when this option is set, the value of the `m_staticRange` variable is set to true.

In addition to the vertical range, also the x-axis range is constantly updated to keep the `m_lastUpdate` timestamp at the far right of the range and the timestamp `(m_lastUpdate - m_range)` at the far left, unless the chart is in scrolling mode, which will be analyzed in a section below. The `m_lastUpdate` variable represents

the timestamp at which the last peak was received, while the `m_range` one represents the timeframe the user has selected to be shown in the chart.

```
void TimeChart::adjustRange(QPair<int, int> sensor, qreal
   peak, qint64 timestamp)
{ ...
    if(m_start.toMSecsSinceEpoch() == 0)
    {
        QDateTime now =
           QDateTime::fromMSecsSinceEpoch(timestamp);
        m_start = now;
        xAxis->setRange(now, now.addMSecs(m_range));
    }
    else if(m_lastUpdate.toMSecsSinceEpoch() >
       xAxis->max().toMSecsSinceEpoch() && !m_scrollingMode)
    {
        xAxis->setRange(m_lastUpdate.addMSecs(-m_range),
           m_lastUpdate);
    }
    ...
    if(peak < m_min.second)
    {
        if(!m_staticRange && !m_scrollingMode)
            yAxis->setMin(peak - PEAK_OFFSET_TIME_CHART);
        m_min = {sensor, peak};
    }
    if(peak > m_max.second)
    {
        if(!m_staticRange && !m_scrollingMode)
            yAxis->setMax(peak + PEAK_OFFSET_TIME_CHART);
        m_max = {sensor, peak};
    }
    updateCrosshairCoord();
}
```

Figure 4.18: *TimeChart::adjustRange*

Crosshair

The functionality implemented in this class allows the user to project on the x and y axes the current coordinates of the mouse position by simply holding the left mouse button. This class was implemented by using the `QGraphicsScene` of

the `BaseChart` and directly calling its paint methods to draw the crosshair overlay composed of `QGraphicsItem` objects.

With the `checkCoordConstraints()` method, the time chart makes sure not to allow the crosshair to be printed outside the charted area but to clip it to its boundaries. By moving the mouse outside the bounds of the chart, while holding down the left mouse button, it is possible to fix one coordinate at an edge of the chart and vary the other.

Every time the `adjustRange()` method is called, also the method `updateCrosshairCoord()` (Figure 4.19) is executed to update the coordinates of the crosshair projected to the axes that, otherwise, would be stuck at the values in which the mouse button was initially pressed without considering the scale change.

```
void TimeChart::updateCrosshairCoord()
{
    auto series = chart()->series()[0];
    auto pos = chart()->mapFromParent(m_mousePosition);
    auto coord = chart()->mapToValue(pos, series);

    coord = checkCoordConstraints(coord, pos);
    m_crosshair.changeCoordinates(coord);
}
```

Figure 4.19: *updateCrosshairCoord function*

Scrolling mode

Since the chart by default always shows the most recent `m_range` time window, a functionality to scroll the chart was designed to allow the user to see data points older than that time frame. To implement that, the following event handlers were overridden:

- `wheelEvent()` (Figure 4.20): this event is fired when the wheel mouse is scrolled; the chart is scrolled by the number of degrees the wheel is rotated. It activates the scroll mode when the user scrolls backward the chart and deactivates it when the chart is scrolled forward to the timestamp of the last received peak.

- `mouseDoubleClickEvent()`: the event is used to detect the double click on the chart area. It deactivates the scroll mode independently by the chart time position and resets the timeframe visualized to display the most recent timestamp.

While the chart is scrolled, the y-axis range is automatically adjusted using the `setMinMax()` function to visualize all the data points in the time range displayed. If the crosshair is active, its coordinates are updated to reflect the change.

```
void TimeChart::wheelEvent(QWheelEvent *event)
{
    qreal degree = event->angleDelta().y() / 8;
    ...
    if(xAxis->min() > m_start && degree < 0)
    {
        m_scrollingMode = true;
        chart()->scroll(degree, 0);
        if(xAxis->min() < m_start)
            xAxis->setMin(m_start);
    }
    else if(xAxis->max() < m_lastUpdate && degree > 0)
    {
        m_scrollingMode = true;
        chart()->scroll(degree, 0);
        if(xAxis->max() > m_lastUpdate)
        {
            xAxis->setMax(m_lastUpdate);
            m_scrollingMode = false;
        }
    }

    setMinMax();
    updateCrosshairCoord();
}
```

Figure 4.20: *wheelEvent implementation*

StrainChart

This class extends the `TimeChart` class to implement the virtual method `toChartValue()` for converting the peak variations to a delta strain, as shown in Figure 4.21. It uses the `saveMeasureParams()` slot to retrieve the sensor configuration inserted by the user and to save them for future measurements through the `SessionSettings::saveStrainSensor` method.

```
qreal toChartValue(QPair<int, int> sensor, qreal peak)
   override
{
    if(m_constants.contains(sensor))
    {
        qreal dW = peak - m_initialPeaks[sensor];
        qreal numerator;
        if(m_tempSensors.contains(sensor))
        {
            auto tempSensor = m_tempSensors[sensor].first;
            if(m_lastPeaks.contains(tempSensor))
            {
                qreal dWT = m_lastPeaks[tempSensor] -
                   m_tempSensors[sensor].second;
                numerator = dW - dWT;
            }
            else numerator = dW;
        }
        else numerator = dW;

        return numerator / m_constants[sensor] +
           m_initialStrains[sensor];
    }
    return 0;
};
```

Figure 4.21: *StrainChart::toChartValue*

TemperatureChart

This class extends the `TimeChart` class to implement the virtual method `toChartValue()` for converting the peak variations to a delta temperature, as shown in Figure 4.22. It uses the `saveMeasureParams()` slot to retrieve the sensor configuration inserted by the user and to save them for future measurements through the `SessionSettings::saveTempSensor` method.

4.6.3 FFTChart

This chart class, which extends the `BaseChart` one, is specifically designed to compute and represent the FFT of the selected signals. To calculate the FFT, every sensor has assigned an instance of the `ffft::FFTReal` class included in the FFTReal library [12], developed by Laurent de Soras.

```
qreal toChartValue(QPair<int, int> sensor, qreal peak)
    override
{
    qreal dW = peak - m_initialPeaks[sensor];
    return dW / m_constants[sensor] + m_initialTemps[sensor];
};
```

Figure 4.22: *TemperatureChart::toChartValue*

Since calculating the FFT requires a precise sampling rate, it is not enough to follow the same approach used in the `TimeChart`. To have more precise samplings, in the `updatePeak()` slot is implemented a *for* loop that takes the time elapsed between a peak received and the following one and divides it by the `m_sampling` value; it then create synthetic samples in-between, at the correct sample rate, considering the value of the signal in that time frame constant (Figure 4.23).

```
void FFTChart::updatePeak(int channel, int grating, qreal
    peak, qint64 timestamp)
{
    ...
    int delta = timestamp - m_lastPeaks[sensor].timestamp;
    for(int i=0; i < delta / m_sampling; i++)
    {
        m_lastPeaks[sensor].timestamp += m_sampling;

        if(m_lastPeaks[sensor].timestamp == timestamp)
            addSample(sensor, peak);
        else
            addSample(sensor, m_lastPeaks[sensor].peak);
    }
    m_lastPeaks[sensor].peak = peak;
}
```

Figure 4.23: *updatePeak slot*

When the selected amount of samplings, saved in the `m_window`, is collected, the FFT is computed by calling the `computeSpectrum()` method (Figure 4.24). The `FFTReal::do_fft()` method returns the output vector of the algorithm (f), which is processed to create the series reflecting the spectrum of the sampled input signal. By default, the item at $f[0]$ is not plotted, as it represents the DC component of

the signal and may throw out of scale the vertical range of the chart. The user can set the `m_plotDC` variable to `true` by acting on the options to change that behavior and plot the DC component.

```
void FFTChart::computeSpectrum(QPair<int,int> sensor)
{ ...
    qreal f[m_window];
    m_fftEngines[sensor]->do_fft(f,
       m_buffers[sensor].constData());

    QList<QPointF> points;
    ...
    for(int i = 0; i<m_window/2; i++)
    {
        qreal abs;
        if(i == 0)
        {
            if(!m_plotDC) continue;
            abs = f[i] / m_window;
        }
        else
        {
            qreal real = f[i];
            qreal img = f[m_window/2 + i];
            abs = qSqrt(real*real + img*img) / m_window;
        }

        qreal freq = i * (1/(m_sampling/1000.0)) / m_window;
        ...
        points.append(QPointF(freq, abs));
    }

    m_series[sensor]->clear();
    m_series[sensor]->append(points);
    ...
}
```

Figure 4.24: *computeSpectrum method*

4.7 SessionLogger

This class is used to save on file the peaks received by the selected sensors; it exposes the slots in Figure 4.25. By calling the `changeSettings()` slot, the sampling frequency (`m_sampling`), duration of the measurement (`m_duration`), and the sensors to be logged are set. When the `start()` slot is called, the actual acquisition starts: a file with the name `<yyyyMMdd-hhmmss.log>`, pointed by `m_file`, is created in the *acquisitions* folder, and the sampling starts. The `stop()` slot can be used to stop the acquisition before the `m_duration` time runs out.

```
class SessionLogger : public QObject
{ ...
public slots:
    void start();
    void stop();
    void changeSettings(long sampling, long duration,
       QList<QPair<int, int>> sensors);
    void updatePeak(int channel, int grating, qreal peak,
       qint64 timestamp);
};
```

Figure 4.25: *SessionLogger slots*

Since the log file takes a snapshot of all the selected sensors at a given timestamp, and the sensor peaks are not received in a synchronized way, the algorithm below (Figure 4.26) was designed in the `updatePeak()` slot, which is connected to the `ConnectionManager::peakReceived()` signal. The `m_samples` buffer starts empty and is the one that is saved recurrently to file, while the `m_peaks` buffer contains the last peak received for every sensor. When a peak with a timestamp equal to the timestamp to sample (`m_sampleTS`) is received, it is saved in the `m_samples` buffer; instead, if the timestamp is higher, the value already present in `m_peaks` is put in the `m_samples` buffer and the newly received value overwrites it. After all the sensors are sampled at `m_sampleTS` or a timestamp higher or equal to `m_sampleTS + m_sampling` is received, the `m_samples` buffer is logged and cleared, and the `m_sampleTS` is increased by `m_sampling`, starting the new sampling cycle. In the latter case, the sensors that have not been sampled yet are considered constant at the last registered peak.

When the `stop()` slot is called, either automatically or manually, the `m_file` is closed.

```
void SessionLogger::writeRow()
{ ...
    m_samples.clear();
    m_sampleTS += m_sampling;
    ...
}

void SessionLogger::addSample(QPair<int,int> sensor, qreal
   peak)
{
    if(m_samples.contains(sensor))
    {
        for(auto s : m_peaks.keys())
        {
            if(!m_samples.contains(s))
                m_samples[s] = m_peaks[s];
        }
        writeRow();
    }
    m_samples[sensor] = peak;
    if(m_samples.count() == m_peaks.count())
        writeRow();
}

void SessionLogger::updatePeak(int channel, int grating,
   qreal peak, qint64 timestamp)
{ ...
    if((timestamp > m_sampleTS &&
       !m_samples.contains(sensor)) || timestamp > m_sampleTS
       + m_sampling)
    {
        addSample(sensor, m_peaks[sensor]);
    }
    else if(timestamp == m_sampleTS || timestamp ==
       m_sampleTS + m_sampling)
    {
        addSample(sensor, peak);
    }
    m_peaks[sensor] = peak;
}
```

Figure 4.26: *Logging algorithm*

4.8 Views

The views are only used to set the layout of the widgets inside them and to define their various signal/slot interconnections without defining any relevant logic, as shown in Figure 4.27. The only exception is the `MeasuresView` class, which will be analyzed in the next section.

4.8.1 MeasuresView

This view shows strain and temperature sensors, and since a temperature may compensate for a strain, a check must be performed before adding one of the first types. To do so, the `ConnectionManager::configReceived()` signal is not directly connected to the `ChartLegend::addSensor` slot, but instead, it is proxied by the `MeasuresView::checkSensor` slot implementing a mechanism to let the strain sensors await for the temperature one they depend on, before adding them to the chart.

When a new sensor configuration is received, the `checkSensor()` method checks if the relative temperature sensor has already been added by looking in the `m_sensors` vector. If it is found, the strain sensor is added to the view; otherwise, it is added to the `m_sensorsQueue`. When a temperature sensor configuration is received, the sensor is immediately added, and all the strain sensors present in the `m_sensorsQueue`, depending on it, are added as well.

4.9 Middleware

The Middleware software was mostly kept as it was implemented in the previous thesis works; it was just modified to support the TCP connection mode. The data sent on the channel is JSON encoded.

The TCP socket server is started with the `start_tcp_server()` method, which opens a socket binding it to the interface `0.0.0.0` on port `TCP_SOCKET_PORT`, defined in the `parser.hpp` header file. After the socket is opened, the `tcp_clients_collector()` thread is started and assigned to the handler `tcp_clients_th`; this thread runs in a loop and accepts new incoming connections, saving the client handlers in the `tcp_clients` vector and sending, as the first message on the channel, the object containing the currently active gratings (Figure 4.28).

Then, when the amount of peak data collected reaches the configured `PEAK_DATA_NUMBER`, along the `mongodbDAO::insertMultipleData()` method, which sends it to the MongoDB database, is also called the function `send_tcp_data()`; this function sends the available data in batch to all the clients that are currently connected to the socket, using the JSON object presented in Figure 4.29.

```
PeakTimeView::PeakTimeView(QWidget *parent)
...
{
    QSplitter *charts = new QSplitter();
    PeakChart *chart1 = new PeakChart();
    TimeChart *chart2 = new TimeChart();;
    ...
    charts->addWidget(chart1);
    charts->addWidget(chart2);
    ...
    ChartLegend *legend = new
       ChartLegend(SessionSettings::PeakTimeView, "Sensors",
       {chart1, chart2});
    connect(&g_connection,
       &ConnectionManager::configReceived, legend,
       &ChartLegend::addSensor);

    QPushButton *options = new QPushButton("Options...");
    connect(options, &QPushButton::clicked, m_optionsDialog,
       &MainOptionsDialog::exec);
    connect(m_optionsDialog,
       &MainOptionsDialog::dynamicRangeSet, chart1,
       &PeakChart::setDynamicRange);
    connect(m_optionsDialog, &MainOptionsDialog::gridSet,
       chart1, &BaseChart::showChartGrid);
    connect(m_optionsDialog,
       &MainOptionsDialog::samplingChanged, chart2,
       &TimeChart::changeSampling);
    ...
    QVBoxLayout *sidebar = new QVBoxLayout();
    sidebar->addWidget(legend);
    sidebar->addWidget(options);
    ...
    QHBoxLayout *layout = new QHBoxLayout(this);
    layout->addLayout(sidebar);
    layout->addWidget(charts);

    this->setLayout(layout);
}
```

Figure 4.27: *PeakTimeView constructor*

```
{
    "active_gratings": [x, y, z, ...]
    // where x, y, and z are the grating indexes
}
```

Figure 4.28: *Active gratings data structure*

```
{
    "index": number,        // grating index
    "timestamp": number     // data read by the Interrogator
    "curr_time": number     // data sent out by the Middleware
    "wavelength": number    // sensor peak
}
```

Figure 4.29: *Peak data structure*

Chapter 5

Tests & results

During the development phase, tests were mainly conducted using an Interrogator emulator; however, during the alpha testing phase, some scenarios were tested with real devices in the DIMEAS laboratory. Below will be described some of the tests performed.

5.1 Instruments

The following devices were used for the test scenarios:

- **Windows laptop**: the FBG Data Analyzer application ran on a Windows 10 64-bit machine equipped with an Intel i7-11800H @ 2.3GHz CPU, 32GB of RAM, and an NVIDIA GeForce RTX 3050 Ti GPU;
- **Raspberry Pi 3 Model B**: the Middleware ran on top of the Raspberry Pi OS v5.15;
- **MongoDB**: a remote MongoDB instance hosted on a server of the Politecnico di Torino reachable from the internet;
- **Interrogator**: SmartScan by SmartFibres (Figure 5.1);
- **Climatic chamber**: the BEGER KK-50 CHLT (Figure 5.3) was used as climatic chamber;
- **Carbon fiber specimen**: a flexible carbon fiber bar (Figure 5.2) used for traction and compression mechanical tests;

5.1.1 Setup

The Raspberry Pi was connected to the Interrogator with an Ethernet cable, while it was connected to the internet thanks to an iPhone 12 used as a hotspot. The

laptop running the FBG Data Analyzer was connected to the same hotspot allowing the two devices to be reachable to one another in the same LAN, enabling the use of the TCP connection mode.

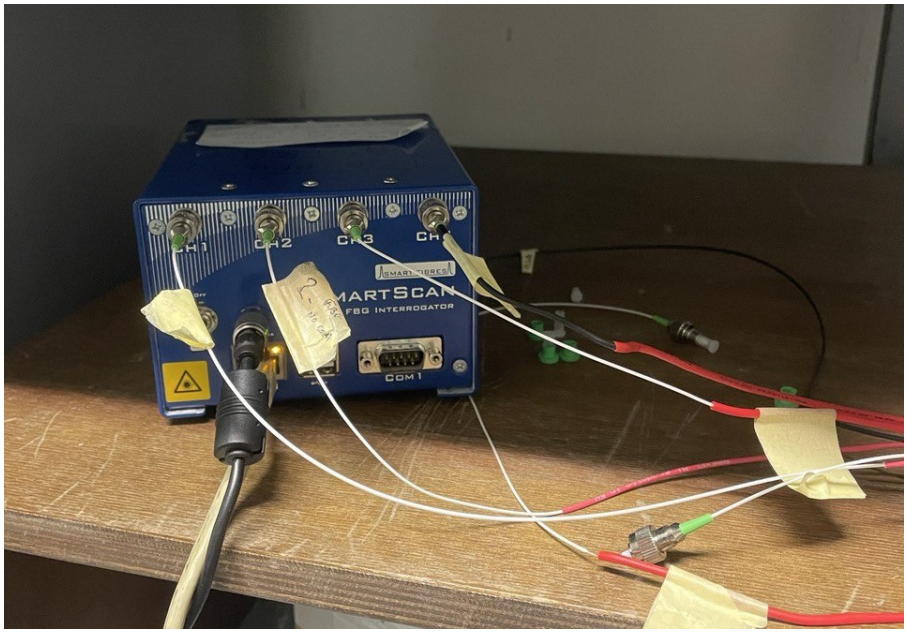

Figure 5.1: *Interrogator*

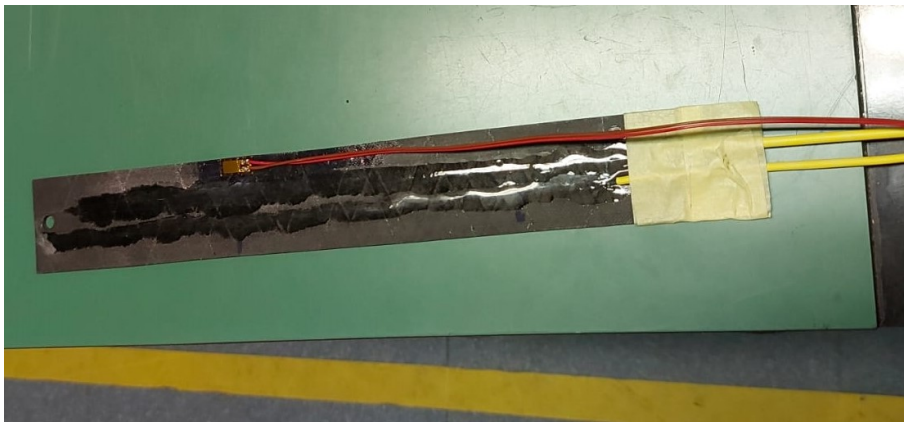

Figure 5.2: *Carbon fiber specimen*

Figure 5.3: *Climatic chamber*

5.2 Temperature correlation validation

The objective of this test was to determine the actual validity of the functionality that allows the visualization of the temperature measured by the FBG. In particular, this is possible by converting the registered wavelength values using the appropriate calibration coefficients previously calculated by the operator.

5.2.1 Procedure

The focus of this test was only the sensor in channel 0, grating 0. The climatic chamber temperature was taken to 100°C stable; then the sensor was configured in the *Measures view* with a k_T of 0.01 and a T_0 of 100.

After the configuration was completed, the climatic chamber temperature control was stopped, and the door opened; as soon as that was done, the sensor immediately began a steep descent to a low of about 50°C, as can be seen in Figure 5.4. The drop after the relatively clean first phase is characterized by a peak up and then a lower rate descent full of noise caused by the turbulences generated by the constant flow of hot and cold air in and out of the climatic chamber.

Figure 5.4: *Temperature drop*

After a couple of minutes, the door was closed again. Because of the still hot metallic surfaces inside the chamber, the temperature started to rise again with a clean exponential move, stabilizing at about 88°C (Figure 5.5).

5.2.2 Results

During the test sequence, the temperatures registered by the FBG Data Analyzer were constantly compared with the ones measured by the internal temperature sensor of the climatic chamber. The results were excellent as not only the two

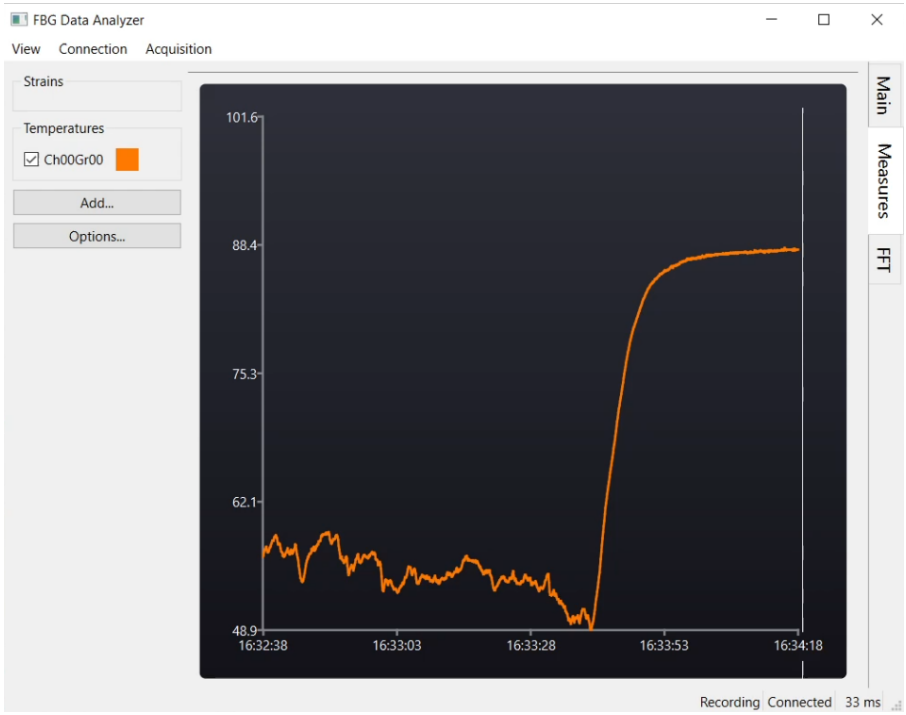

Figure 5.5: *Temperature rise*

temperatures were aligned with a difference of about a degree, but the results also highlighted the extreme reactiveness of the FBG sensor compared to the embedded sensor, thanks to the real-time visualization and synchronization with the physical phenomena.

5.3 Strain correlation validation

The objective of this test sequence was to validate the response of the measured strain gauge to the physical force applied to the carbon fiber specimen.

5.3.1 Procedure

The carbon fiber specimen was positioned to have one extremity supported on the table, fixed, and the other hanging outside the plane with no support. This allowed using the table as a pivot point to band up or down the bar.

With the bar put in place, the FBG sensor in channel 1, grating 0, was configured in the *Measures view* with a k_ε of 0.001 and calibrated with a ε_0 of 0.

With the setup completed, the carbon fiber bar was bent manually and with some sample weights, observing the measured strain gauges. A negative strain was

measured when the bar was banded upward; on the other hand, a positive strain was detected when the bar was bent downward. In addition to the sign of the measured strain, also the magnitude was commensurated to the amount of force applied, returning to the base value of 0 when no forces were involved.

In the strain chart shown in the upper section of Figure 5.6, it is possible to observe an extract of a measurement generated during the test phase.

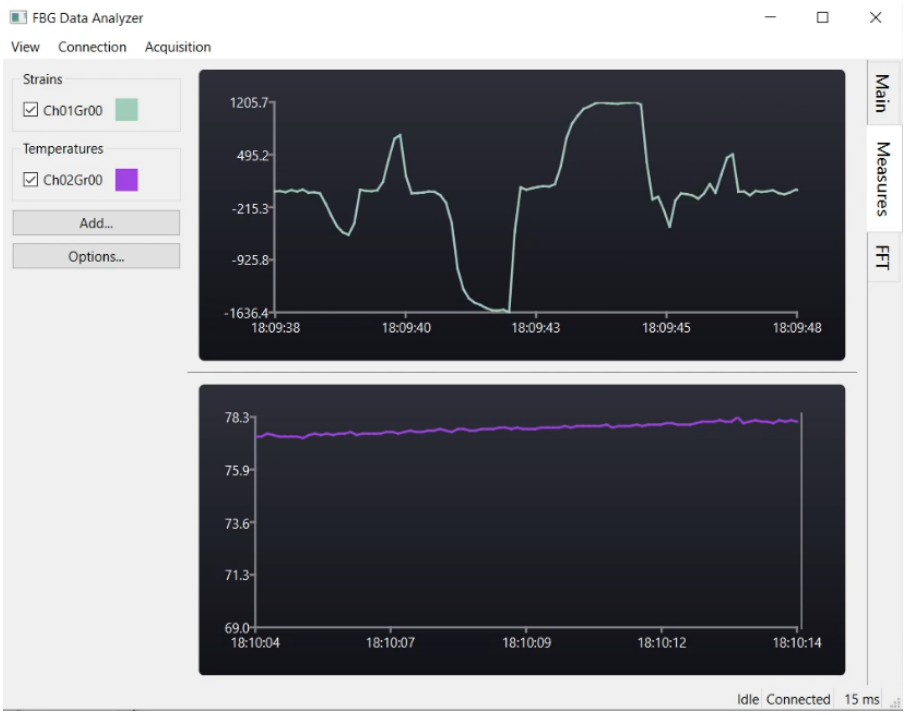

Figure 5.6: *Strain chart trace*

5.3.2 Results

The feature implementation returned positive results, allowing the operator to precisely and efficiently visualize in the chart the strain applied to the object in the unit of measure determined by the k_ε and ε_0 values.

5.4 Multiple sensors accuracy

This test aimed to perform a measurement with two sensors to verify and compare the measured values with the expected ones and to verify the legibility of multiple parallel series on the *Measures view* charts.

5.4.1 Procedure

The climatic chamber was taken to 70°C; at this point, the two sensors at channel 0, grating 0, and channel 2, grating 0, were both configured with a k_T of 0.012 and calibrated at the current temperature. Then, the climatic chamber was configured to lower the temperature to 50°C, and the chart was observed for the next 10 minutes until the chamber reached the set temperatures.

As it is possible to observe in Figure 5.7, the gradual descent in temperature was different for the two sensors. The *Ch0Gr0* slope was characterized by quite some noise; this behavior was expected as the optical fiber containing the sensor was uncoated and kept suspended by two clips with more freedom of movement in comparison to the *Ch2Gr0*, which instead was attached on top of a bar, covered in resin. This is the reason that made the descent of the *Ch2Gr0* sensor very steady.

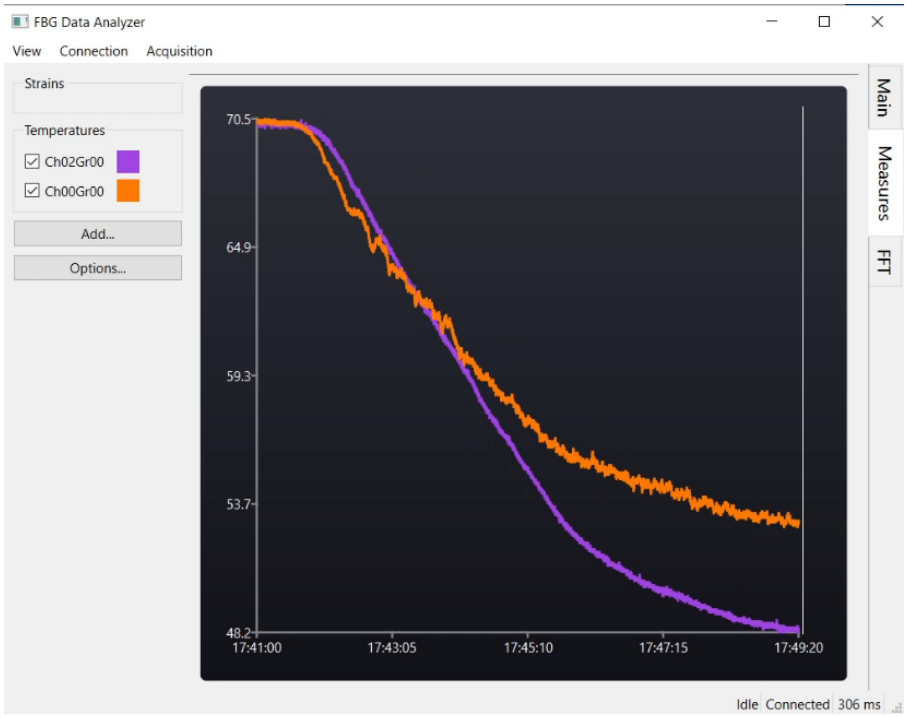

Figure 5.7: *Trace of two temperature sensors*

From the chart it can be noticed a difference of about 4°C between the readings of the two sensors; this is in part expected, since the *Ch0Gr0* sensor (hotter) is positioned about 20cm above the *Ch2Gr0* one (colder), and the embedded temperature sensor of the climatic chamber is placed in its upper section, closer to the *Ch0Gr0*; however, the difference could have also been partially exaggerated by the imprecision of the constant used to configure the *Ch2Gr0* sensor.

5.4.2 Results

Both the traces were clearly visible and analyzable, allowing the user to study the phenomena with great detail. They highlighted how the FBG Data Analyzer could be a great instrument to study events measured with the FBG sensors, validating the premises of this thesis.

5.5 Performance analysis

During the tests described above, latency, memory usage, and CPU usage were constantly monitored to analyze the characteristics and performances of the FBG Data Analyzer.

5.5.1 Latency

The latency was monitored thanks to the built-in latency indicator present in the status bar of the application. The latency was calculated from the moment the Middleware elaborated the data to the one where the FBG Data Analyzer received it.

TCP connection

During the tests described in Section 5.2 and Section 5.3, the connection mode selected was the TCP one. During the tests, the minimum latency registered was 14 ms, while the maximum was recorded at 66 ms. The overall average latency during the measurements was around 33 ms.

Database connection

The test described in Section 5.4 was performed using the database connection mode. During this scenario, the minimum detected latency was 118 ms, while the maximum value registered was 790 ms. During the timeframe of the test, the average computed latency was 246 ms.

Results

As expected, the TCP connection method was the fastest in delivering the data from the Middleware to the FBG Data Analyzer; therefore, it is the most indicated for scenarios requiring real-time responses to the measured physical events. For every other situation, using one method over the other is not a real game changer since the database connection method has latencies more than acceptable for typical measurement campaigns.

5.5.2 CPU & RAM usage

The CPU and memory statistics were monitored utilizing the *Process Explorer* [13] application, distributed by Microsoft. The application lists all the processes running on the machine and, above other things, allows the user to open a chart displaying the CPU, memory, and I/O usage of a selected process. The monitoring was made over the course of a testing day, allowing multiple samples of different scenarios to be gathered; that data was used to make the conclusions below.

CPU

As can be seen from Figure 5.8 and Figure 5.9, the FBG Data Analyzer is very lightweight on the CPU. In a standard measurement scenario, its usage bounced from 1% to 3%, with peaks over these values, up to about 7%, when scrolling charts containing different series and large time-windows; this increased CPU usage is caused by the computation of new minimum and maximum values, which must be continuously updated to adjust the chart range to the displayed data.

Memory

The memory was monitored to detect eventual memory leaks or general anomalies in the use made of the memory by the FBG Data Analyzer. After multiple tests have been executed, it can be said with certainty that the application is not affected by any significant memory issues.

To demonstrate a typical memory consumption trace, a standard execution was monitored for about 30 minutes. Figure 5.8 shows the first 10 minutes of monitoring, where the program instantiates new objects and fills the memory with new data points incoming from the Middleware; in this phase, memory usage slowly increases, remaining at acceptable levels. The FBG Data Analyzer was kept running for another 20 minutes while the data received was also being logged on the local disk; a snapshot of the last 10 minutes of execution has been impressed in Figure 5.9. As can be noticed, the memory utilization at this point remained constant, demonstrating that the series data point cap was working correctly; indeed, at that moment, all the series had reached the `RETENTION_COUNT` points limit, at which the data structure becomes a FIFO (First In First Out), deleting one element for each new one added to the buffer.

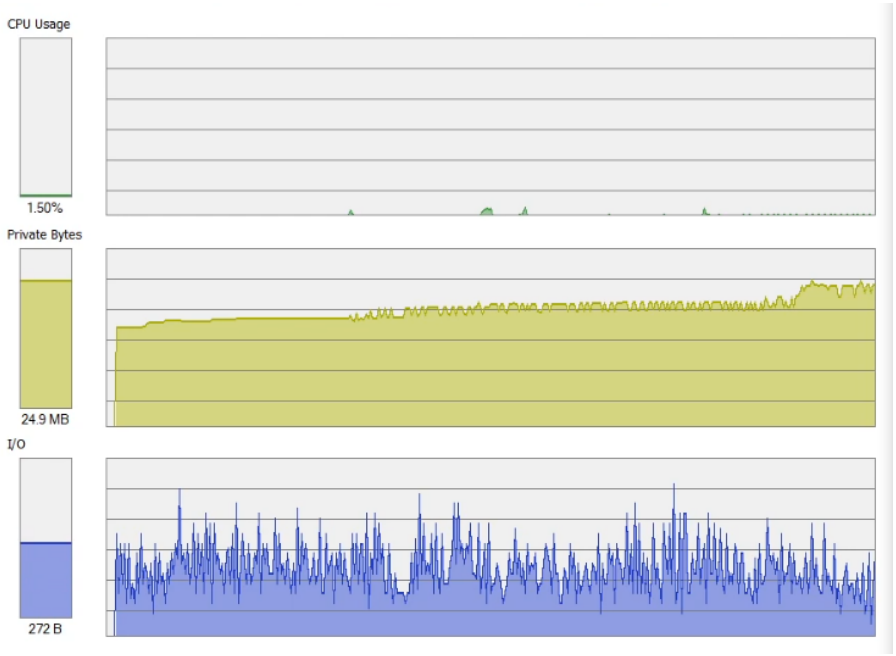

Figure 5.8: *CPU and RAM traces of the first 10 minutes of execution*

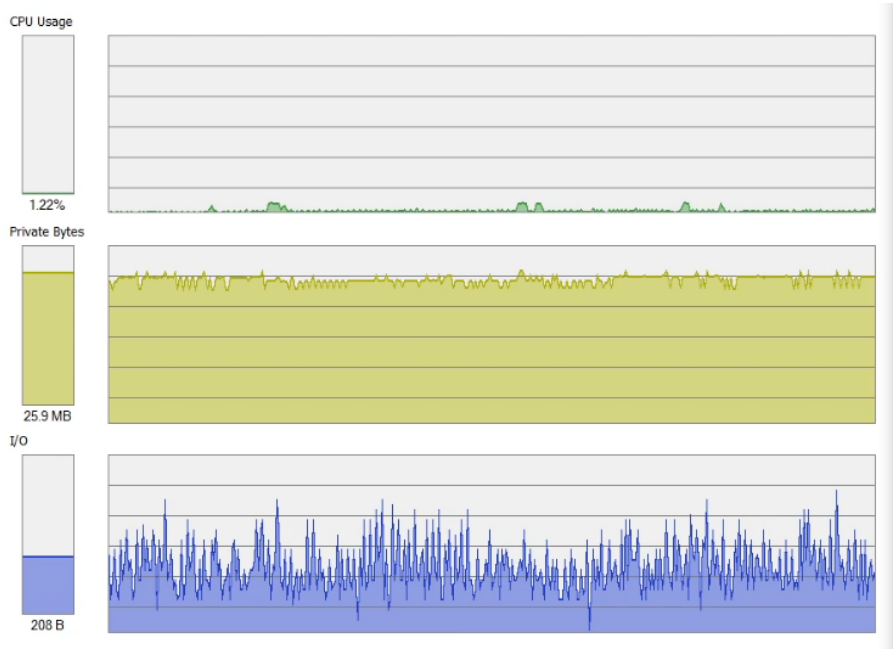

Figure 5.9: *CPU and RAM traces of the last 10 minutes of execution*

Chapter 6

Conclusions

The objective of this thesis work was to create an innovative visualization tool to allow the researcher of the PhotoNext team to have an additional instrument for studying and validating the properties of the FBG sensors.

The developed application was designed on top of a thorough requirements analysis and custom-tailored for the operators that will use it. The development phase was also preceded by a study of the technologies used in the context of this project and a study to choose the best suiting stack for implementing the proposed solution.

6.1 Results

The FBG Data Analyzer application, the result of the efforts put into this project, was developed with a modular architecture designed on top of the Qt framework, making it easily expandable with new features, as it will be required with the advancement of research in the field.

At the current state, the application meets the initial requirements, as it can visualize in a user-friendly interface the instantaneous sensor peaks, their trend over time, their correlation with the physical variations causing the peak changes, and the spectrum of the signals received. On top of that, the TCP connection mode was successfully introduced, and a logging system was implemented to save the measurements on the local disk, allowing the user to easily import them into post-analysis software.

All the features mentioned above underwent multiple tests during the development phase and were validated with the final tests performed in the laboratory, with the collaboration of one of the researchers of the PhotoNext group. During the tests, in addition to the proper functioning of the implemented features, it was also possible to verify how the response times of the FBG Data Analyzer remained at acceptable levels for a real-time application, one of the key requirements that

emerged from the initial analysis.

The application in its current state was shown and tested with some stakeholders, receiving positive feedback.

6.2 Future works

All the source code of the FBG Data Analyzer and the Middleware is available in a Bitbucket repository owned by the DAUIN. Thanks to the designed architecture, it should be reasonably easy to implement new features and maintain the current codebase.

The project is still in the beta stage; therefore, there are surely bugs to be discovered and fixed, and additional tests to be carried out.

In the current phase, while a measurement is ongoing, the application partially supports variation to the physical layout of the sensors; still, research and development of a system to enable this fully and securely are encouraged.

All the resources and essential elements to support multiple Interrogators to be used in the same application instance are there, and they just need to be put together; instead, the opposite case is already supported, allowing multiple FBG Data Analyzer instances, also running in different systems, to receive the data of a single Interrogator.

Finally, the porting to another operating system could be worth the effort, requiring a fraction of the work that would be needed if the application was not implemented on top of the Qt platform.

Bibliography

[1] Politecnico di Torino. Photonext. https://www.photonext.polito.it/.

[2] Infibra Technologies. Fbg overview. http://www.infibratechnologies.com/technologies/fiber-bragg-gratings.html.

[3] HBM. What is a fiber bragg grating? https://www.hbm.com/en/4596/what-is-a-fiber-bragg-grating/.

[4] Politecnico di Torino. ICARUS. https://icarus.polito.it/.

[5] BEGER. Climatic chamber kk-50 chlt. https://beger.si/en/product/kk-50-chlt/.

[6] Smatfibres. Smartscan. https://www.smartfibres.com/products/smartscan.

[7] Raspberry Pi. Raspberry pi 3 model b. https://www.raspberrypi.com/products/raspberry-pi-3-model-b/.

[8] MongoDB. Mongodb. https://www.mongodb.com.

[9] Qt Group. Qt framework. https://www.qt.io/product/framework.

[10] Qt Group. Signals & slots. https://doc.qt.io/qt-6/signalsandslots.html.

[11] MongoDB. Change streams. https://www.mongodb.com/docs/manual/changeStreams/.

[12] Laurent de Soras. Fftreal. http://ldesoras.free.fr/prod.html#src_audio.

[13] Microsoft. Process explorer. https://learn.microsoft.com/en-us/sysinternals/downloads/process-explorer.