MIDDLE EAST TECHNICAL UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

SUMMER PRACTICE REPORT

CENG 400

STUDENT NAME : Onur ŞİMŞEK

ORGANIZATION NAME : Tübitak Bilgem ILTAREN

ADDRESS : Ümitköy/ANKARA

START DATE : 20 August 2020

END DATE : 17 September 2020

TOTAL WORKING DAYS : 20

STUDENT'S SIGNATURE **ORGANIZATION APPROVAL**

TABLE OF CONTENTS

TABLE OF FIGURES

1. INTRODUCTION

This report document includes what I have done during my internship at Tübitak Bilgem ILTAREN which is located in Ankara including the difficulties that I faced and how I solved them. During my internship, software engineer Deniz Can YILDIRIM was my mentor. In this report, you can find what my project is about and the technical details of my project.

My internship was in between 20.08.2020 and 17.09.2020. This was not my first internship experience, so my aim for this internship was to have some knowledge and experience with concepts such as threading, multithreading, message queues and network protocols.

In Tübitak Bilgem ILTAREN, I have responsibilities in a project of the company in which I try to measure the execution time of the desired functions. All of the internship has been done face to face. Therefore, I had some opportunity to meet with my mentor to discuss what I have done and what I will be doing during the following task.

2. INFORMATION ABOUT PROJECT

My project is called the Execution Timer Application. It is a project where we are trying to measure the execution of a function by using the concepts such as multithreading and message queues.

In most of the conditions, we as engineers want to know how much time has been passed in order for a function to be executed. The reason that we want to know this is that we are trying to fix a bug in our code, or more importantly we are trying to optimize the performance of our program since a program should meet with some non functional and functional requirements. For example, if we can recognize that a function desires much time to execute then we can understand that there is something missing or wrong in our implementation that prevents our function from responding, and we need to optimize it.

In my internship, first I need to understand how I can use threading and message queues and why they are useful. So, at the very beginning of the project, I have tried to learn what is threading, why it is important, and how we can make use of it. After this period of time, I implement my project with only one thread that is for our class. Then, I have modified my project such that it responds to all of the functions from their threads. That is to say, we are calling our functions to measure the execution time of a function by using threads that belong to some different classes. By doing so, I have used multithreading. Finally, these results can be shown in an interface just like a table. To perform this, I dealt with TCP network protocol and applied this to my project.

This is the project that is created in a Linux machine and will be used in a Linux based machine.

2.1 ANALYSIS PHASE

While I was creating the Execution Timer Application, I have used many tools and libraries. The most important ones are Visual Studio, Eclipse, Qt Creator, POSIX interface. Here, you can find brief information about these tools and how I have used them.

2.1.1 Visual Studio

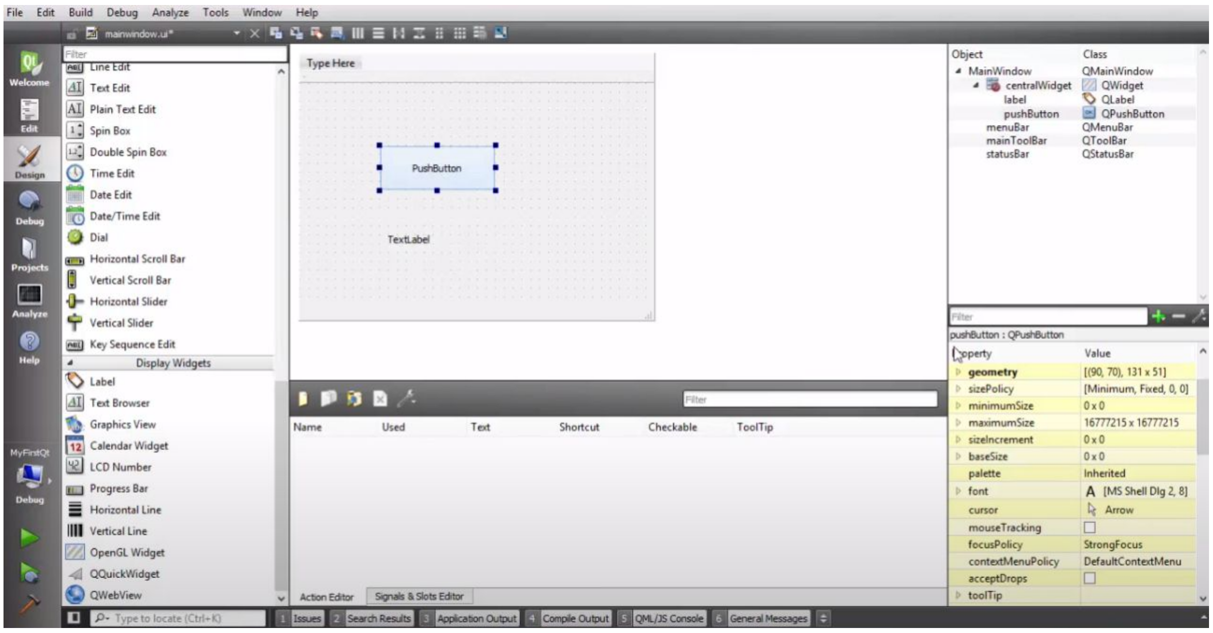
I used Visual Studio for the basic C++ codes of my project that doesn’t require any debugging. I used the terminal of Visual Studio to test these codes with g++ compiler linked with -pthread library and -rt library.

2.1.2 Eclipse

Eclipse is just a code editor that enables you to run or debug your program by using even one button. So, if you are not sure about the cause of a problem such as segmentation fault, you can simply set some breakpoints in your program, and run your program in the debug mode to understand the cause of the problem. I have used Eclipse in my project for this purpose.

2.1.3 Qt Creator

In my project, I had to design an interface by using C++. To achieve this, after some search about GUI programming, I found QT Creator. Qt Creator allows you to design your interface, and add some event to your interface. It may be useless to have a very good looking interface if we fail to add them some functionality. For example, in my interface, I had a button called as *show* with which I can show all of the execution results of the functions.



Object	Class
MainWindow	QMainWindow
centralWidget	QWidget
label	QLabel
pushButton	QPushButton
menuBar	QMenuBar
mainToolBar	QToolBar
statusBar	QStatusBar

Filter

pushButton : QPushButton

Property	Value
geometry	[(90, 70), 131 x 51]
sizePolicy	[Minimum, Fixed, 0, 0]
minimumSize	0 x 0
maximumSize	16777215 x 16777215
sizeIncrement	0 x 0
baseSize	0 x 0
palette	Inherited
font	A [MS Shell Dlg 2, 8]
cursor	Arrow
mouseTracking	☐
focusPolicy	StrongFocus
contextMenuPolicy	DefaultContextMenu
acceptDrops	☐
toolTip	

Figure 1: The interface of Qt Creator

2.1.4 POSIX Interface Library

This library provides a lot of functions with which you can create a thread, create a message queue, send and receive a message. In my project, all of these jobs are done with the help of this POSIX library.

2.2 DESIGN PHASE

The design phase of our project can be observed in two parts, namely the interface design and the backend design.

2.2.1 Interface Part

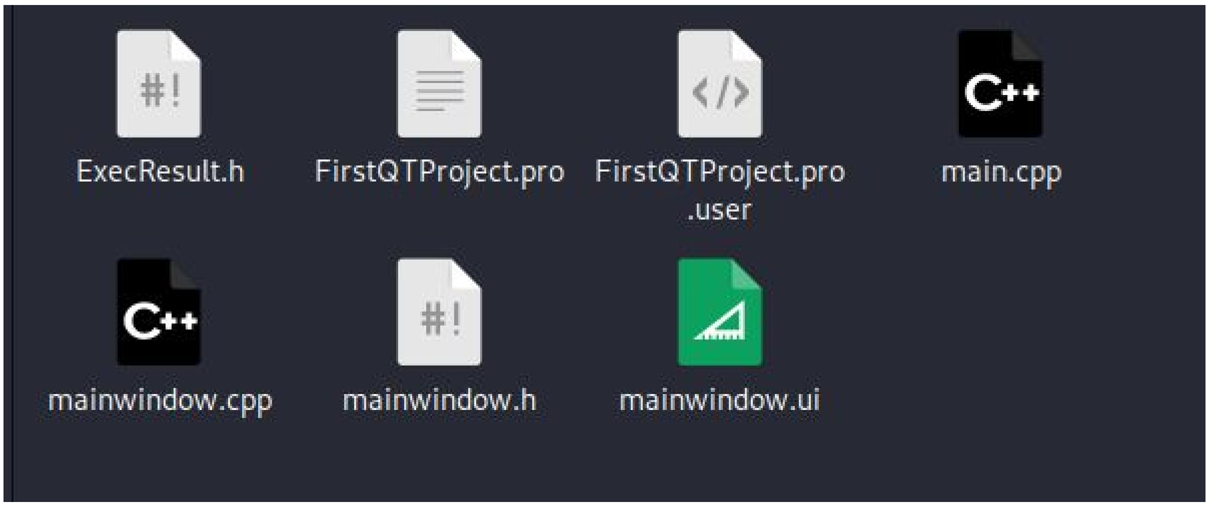


Figure 2: The design of the interface part

Our aim for this project was not to deal with the interface of this project. Rather, we wanted an interface to ease our job while we are displaying results on the screen. So, the interface is not the main focus of our project. Therefore, I dealt with the interface just to have some buttons and results that will be shown when these buttons are pressed. As you can see, there is some part called the ui part in which we are trying to customize the viewing of our project, and also there is a part for the implementation which needs to be done to give some functionality to the buttons.

Figure 2 shows the parts that involve the interface design of the project. This is a part of our project which is a part of QT Creator project.The files except *ExecResult.h* file, is created by using QT Creator. The most important files are :

- **mainwindow.ui:** This is a file where we can drag and drop some buttons or other tools related to the view to the screen. There should be three buttons to listen to the client, show the results and clear the results table.
- **mainwindow.cpp:** This is a part where I implement the server side of the project. The functionalities of the buttons are given in this implementation.
- **ExecResult.h:** This header file should declare a struct which defines the execution results with its fields. In other words, there should be the function

name and the execution time fields which will be used to show the execution time of the function.

I want all of the results to be shown as a table which has two columns as function name and execution time. This should be done by using tableWidget of QT Creator. After briefly talking about the design of the interface, let me dive into the design of the backend part.

2.2.2 Backend Part

This backend part will be the part where we mostly spend time and focus on. I will talk about the design part of the project and why we have such a design. Because in most of the cases designing of the algorithm or the project might be more difficult than implementing it, I think it will be beneficial to talk about the reasons behind the scene.

In our design, we should have a class where we implement the execution timer. In addition, we also need to have more than two classes to test our execution timer class. Eventually, when we are done with the designing of all classes, we should create a static library for our Execution Timer which will be used later on when wanted.

If we look at the structure, we have a header file which includes the declaration of the variables and methods and we have a .cpp file which includes the definition and initializations of the variables. Also, all of the coding is done with the object oriented programming paradigm. That is, everything is inside of a class.

The backend part is not designed all at once. Here is the improvement or the evolution of the backend part.

In the first phase, we should create our exact timer class and methods inside it. This ExecTimer should have a thread. In other words, when we create an object of this ExecTimer class, we are basically starting the thread of the ExecTimer class. And the method calls are running inside of this thread. Now, in this phase, if we think about it, we have two threads. One of them is the main thread and another is the ExecTimer thread. Why are we doing this? Because we want our program not to be crushed by anything. Even one function in one class is problematic, we want our execution to continue. Another reason is the execution time. One can easily create two programs, do some jobs with and without using thread and see the difference between the execution time. If all of your program is created using one thread, then you will need to wait for another task. For the second job to start, you need to wait for the first job to finish. On the other hand, if you use threading, one task is not waiting for another task to finish. That is to say, while we are doing the first job, the execution of the second job is also starting. Therefore, the execution of the first job and the second job is done in parallel. Figure 3 explains all of these.

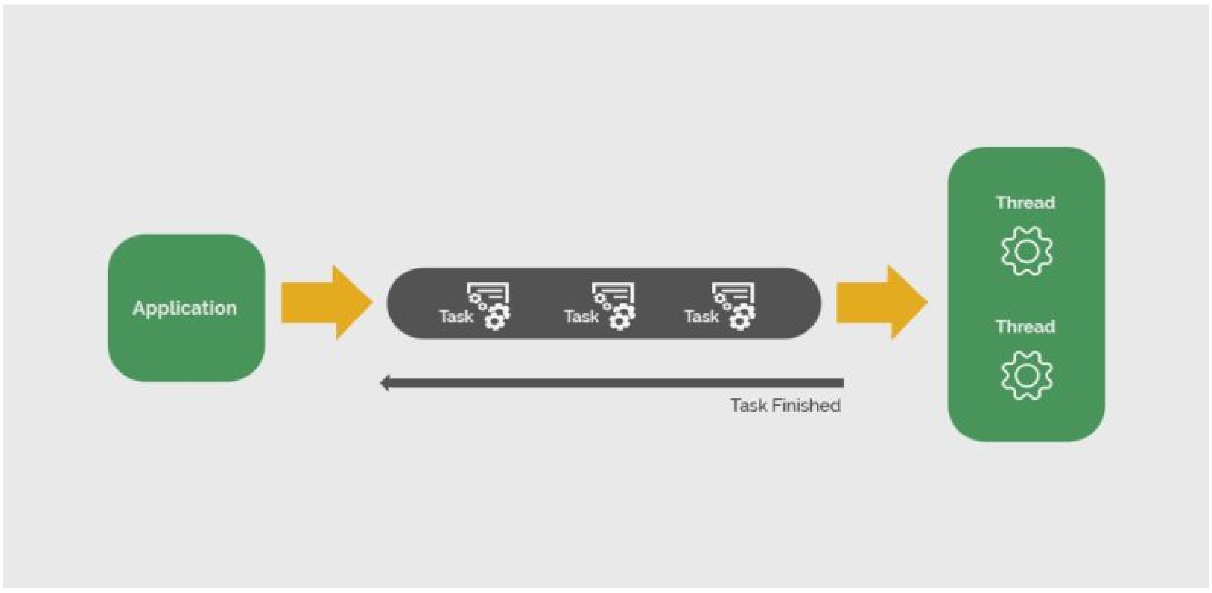


Figure 3: How threading works?

In the second phase, we think that can we do this multithreading? In the company, they may want to call the methods of ExecTimer class within their threads.So, we need to figure out whether we can do the same job with one more class that includes the thread inside it.

In the third phase, I increased the number of classes and did the same job again. Here, my aim is to see how many threads we are going to have and how they are performing in parallel. Also, I had a chance to see how multithreading can be performed.

In the last phase, we saw the results of the execution in the terminal. However, there might be many threads each of which calling a different function.So, why not showing all of the results in the interface as a table which contains the function name and execution time. This had a cost, and will require a knowledge of network protocol. In this project, I prefer to use TCP network protocol. Here, you can see brief reasoning about why I prefer TCP over UDP protocol:

- TCP is connection oriented protocol. So, if you are using TCP network protocol, you have to be sure that connection has been established between client and server.
- TCP is reliable as it guarantees the delivery of the data to the destination.

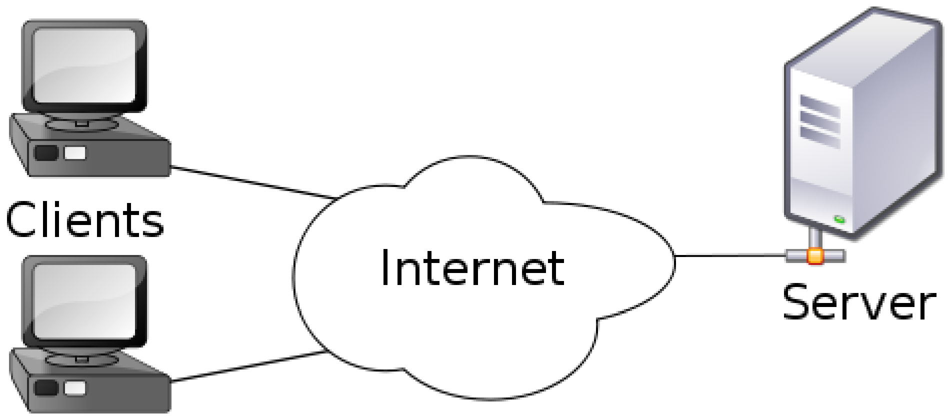


Figure 4: The relationship between clients and server

After the last phase, we moved our implementation to what is shown in Figure 4. You can see how the clients, in which our results are stored, send data to the server which will receive the data. Here, the server is invoked with the help of a button in our interface. When the *LISTEN* button is clicked on the interface, the server should listen to the client. Likewise when the *SHOW* button is clicked on the interface, all of the results should be shown in the interface.

To see all of the execution results, before the client sends data, we should make the server listen to the clients and then we should run the client.

In Figure 5 and Figure 6, you can see how the interface looks like and the effect of running the program on the interface.

The table in Figure 5 is expanding dynamically, in other words the number of rows in the table is incrementing as we keep adding items in the table. The row number appearing in Figure 6, is shown when there are items in the table. There is also a scroll view with which you can see all of the execution results of the functions with columns function name and execution time.

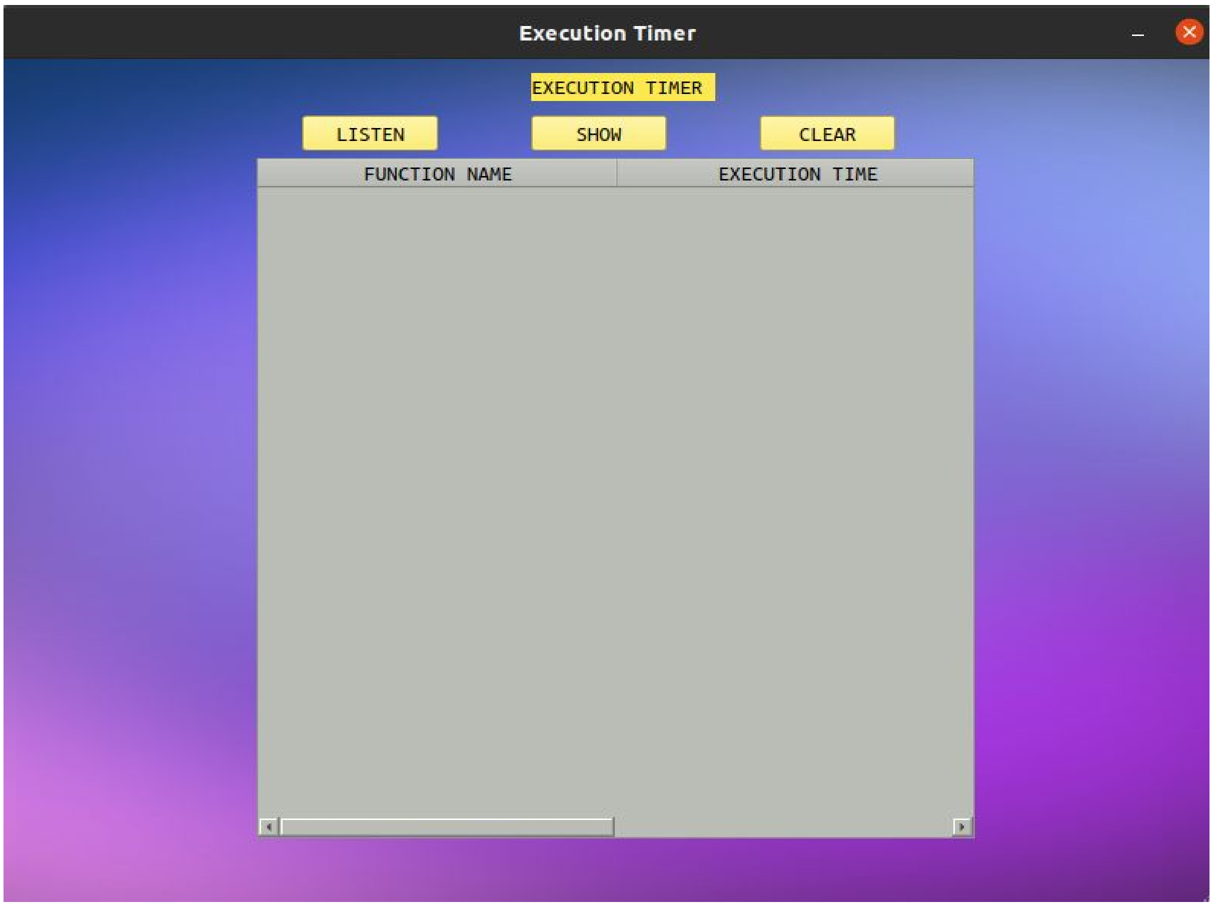
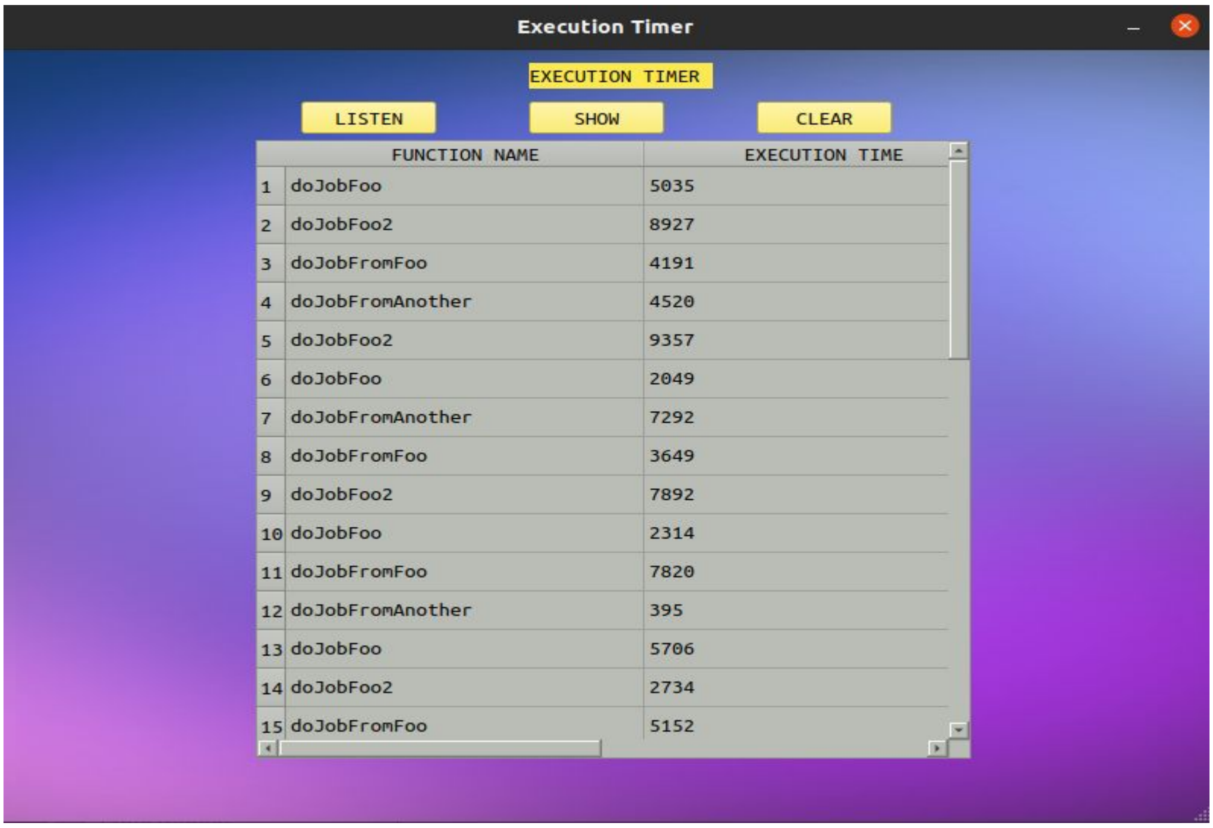


Figure 5: The interface while listening the client



	FUNCTION NAME	EXECUTION TIME
1	doJobFoo	5035
2	doJobFoo2	8927
3	doJobFromFoo	4191
4	doJobFromAnother	4520
5	doJobFoo2	9357
6	doJobFoo	2049
7	doJobFromAnother	7292
8	doJobFromFoo	3649
9	doJobFoo2	7892
10	doJobFoo	2314
11	doJobFromFoo	7820
12	doJobFromAnother	395
13	doJobFoo	5706
14	doJobFoo2	2734
15	doJobFromFoo	5152

Figure 6: The interface after *SHOW* button is clicked

As it can be seen from Figure 6, all of the results are shown in the form of function name and the execution time. If you remember, there was a header file which has ExecResult struct definition in it which has these fields.

2.3 IMPLEMENTATION PHASE

Similar to the design phase, the implementation phase of our project can be observed in two parts, namely the interface implementation and the backend implementation.

2.3.1 Interface Part

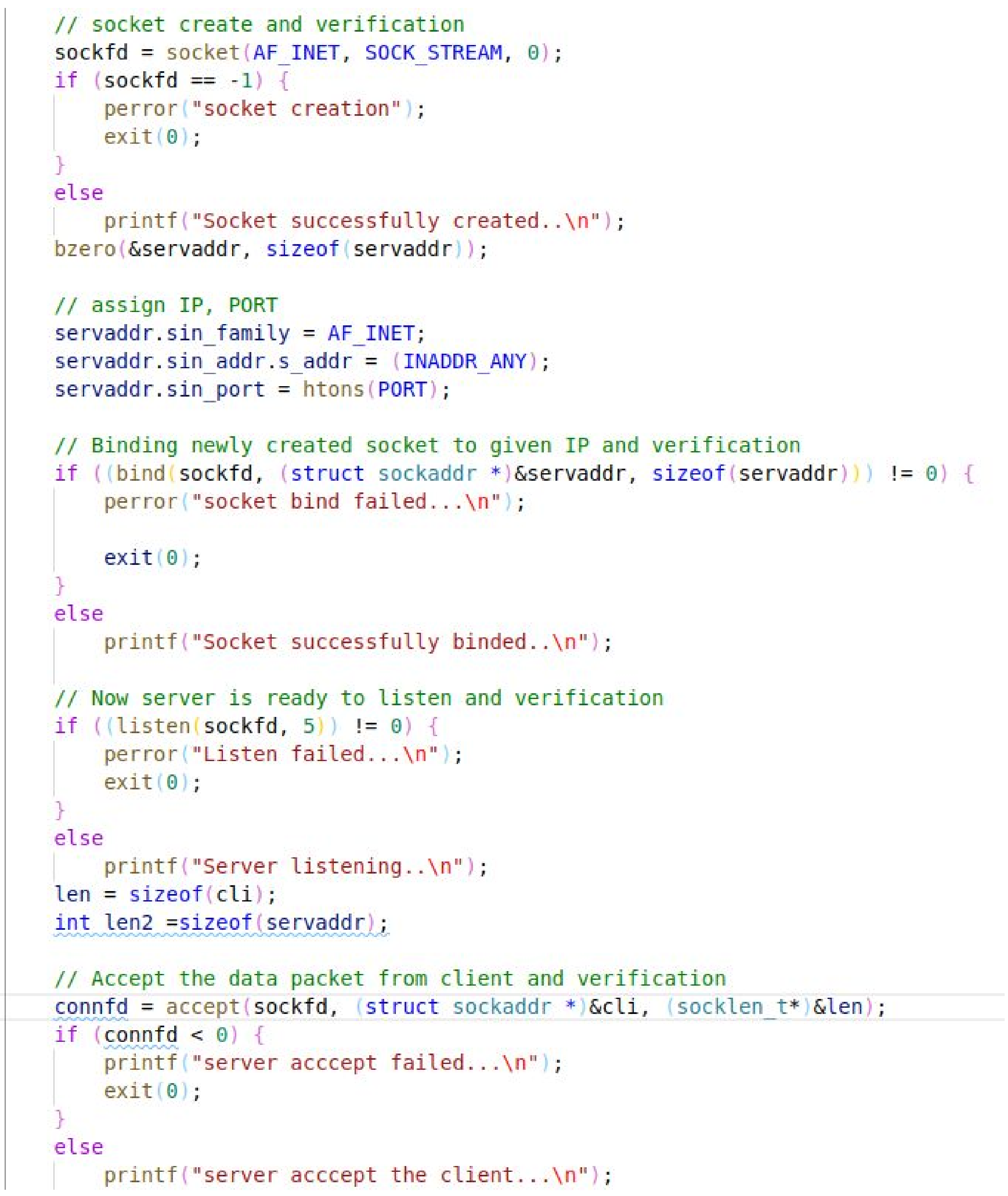
I have divided the implementation part of the interface into two parts. The first part is about the action that should be taken when the *LISTEN* button is clicked. The second part is the action when the *SHOW* button is clicked.

When we click the *LISTEN* button, the handler function of this button will be responsible for:

- Creating the socket
- Binding the socket
- Listening the client part
- Accepting the data packet from client

All of those can be achieved with the help of built in functions with some proper arguments. The most difficult part of working with some built in function is that you give the proper type such as struct to your function. However, your function expects that kind of struct with some proper initial value. This is what I focus on.

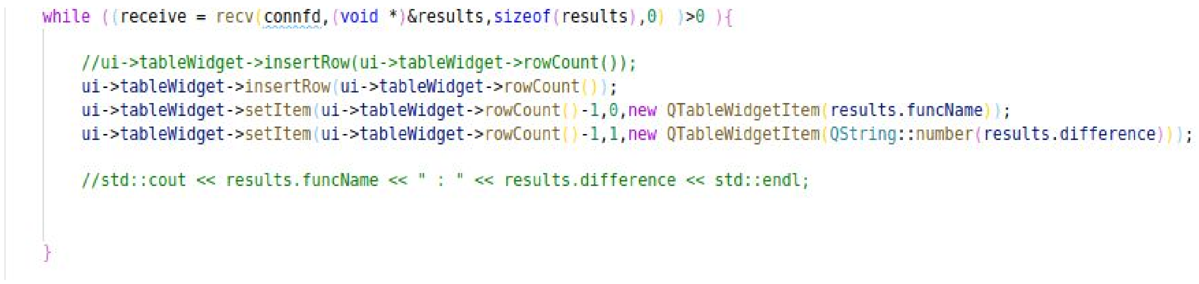
As you will see from Figure 7, all of the functions have a return value. This return value can be used for debugging purposes. For example, if the accept method returns a negative value, this can be caused both by the accept or some other method before the accept method. Therefore, we need to be sure that each function is working properly by looking at the return value of these functions in a step by step manner. The proper return values of the functions can be observed from the manual page of the functions.



```
    // socket create and verification
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        perror("socket creation");
        exit(0);
    }
    else
        printf("Socket successfully created..\n");
    bzero(&servaddr, sizeof(servaddr));

    // assign IP, PORT
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = (INADDR_ANY);
    servaddr.sin_port = htons(PORT);

    // Binding newly created socket to given IP and verification
    if ((bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr))) != 0) {
        perror("socket bind failed...\n");

        exit(0);
    }
    else
        printf("Socket successfully binded..\n");

    // Now server is ready to listen and verification
    if ((listen(sockfd, 5)) != 0) {
        perror("Listen failed...\n");
        exit(0);
    }
    else
        printf("Server listening..\n");
    len = sizeof(cli);
    int len2 =sizeof(servaddr);

    // Accept the data packet from client and verification
    connfd = accept(sockfd, (struct sockaddr *)&cli, (socklen_t*)&len);
    if (connfd < 0) {
        printf("server acccept failed...\n");
        exit(0);
    }
    else
        printf("server acccept the client...\n");
```

Figure 7: The implementation of the server part

When we click the *SHOW* button, the handler function of this button will be responsible for receiving the data coming from the client and showing these results as a table. In Figure 7, you will see a description of how I implement this. We should recognize that we are doing this receiving data job as long as there is data. If there is no data remaining, *recv* will return a negative value and we will be done with the while loop. In a sense, we are receiving as much data as we have sent.



```
while ((receive = recv(connfd,(void *)&results,sizeof(results),0) )>0 ){

    //ui->tableWidget->insertRow(ui->tableWidget->rowCount());
    ui->tableWidget->insertRow(ui->tableWidget->rowCount());
    ui->tableWidget->setItem(ui->tableWidget->rowCount()-1,0,new QTableWidgetItem(results.funcName));
    ui->tableWidget->setItem(ui->tableWidget->rowCount()-1,1,new QTableWidgetItem(QString::number(results.difference)));

    //std::cout << results.funcName << " : " << results.difference << std::endl;


}
```

Figure 8: Receiving the data from the client and showing them in the table

2.3.2 Backend Part

In the implementation of the backend part, we have ExecTimer class and other classes which use the methods of ExecTimer class to measure the execution time of a function. I will focus on the implementation of ExecTimer class and some important parts of the other class.

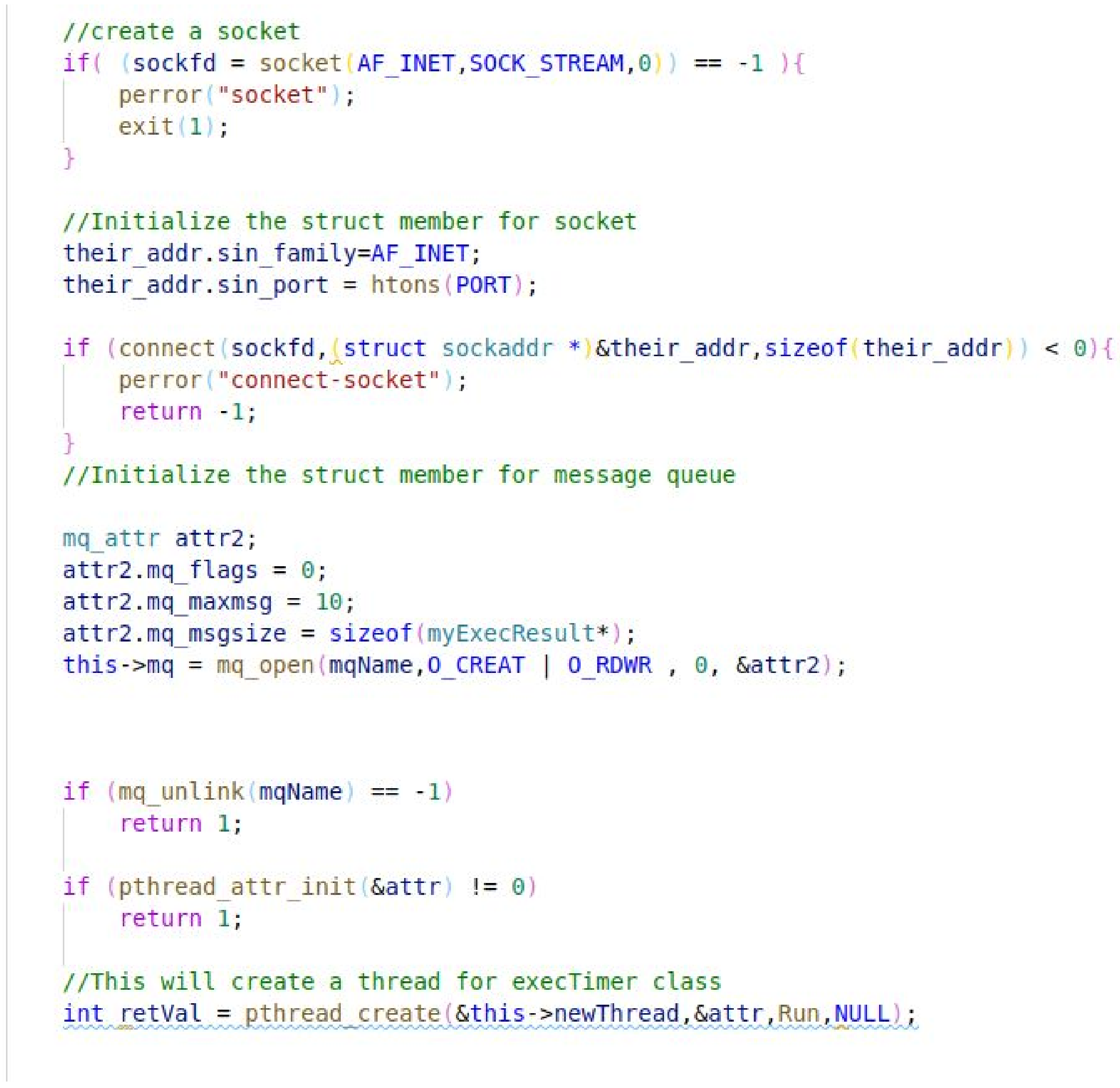
ExecTimer is just like a client where we produce the data which are the execution results in our case and we send the data to the server. Related to ExecTimer class, here is all of the jobs that we have done:

- Opening a message queue
- Sending a message via message queue
- Receiving the message that is sent by message queue
- Closing the message queue
- Creating a new thread
- Send message to the server

I will talk about the implementation of these tasks. There is nothing inside the constructor of ExecTimer class.

In the ExecTimer class, there are some important functions where the above items happen. These are:

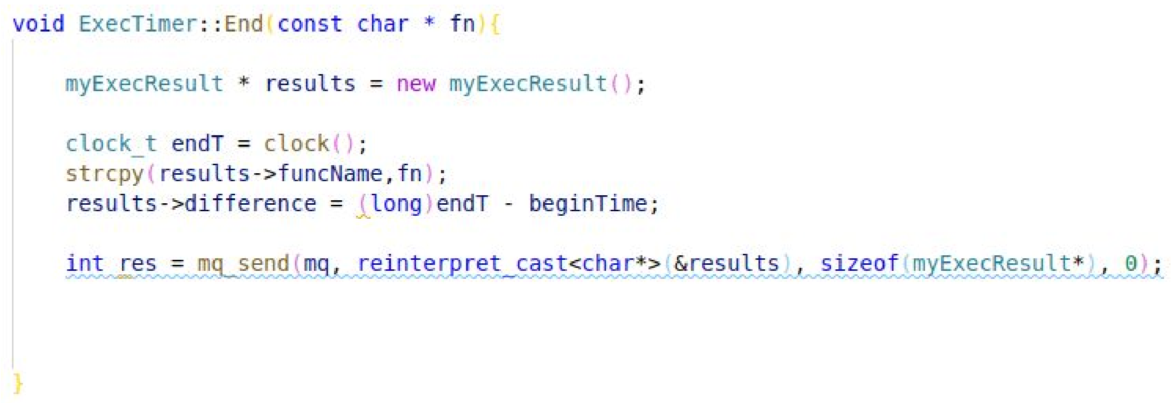
- **Startup:** This function is just like a constructor of our *ExecTimer* class. It is doing all of the important creations or initializations. Creating a socket which we will be using to send or receive data between our program and the interface happens in here. Opening a connection socket, opening a message queue, creating a new *ExecTimer* thread are also done here. That means whenever we call this function of ExecTimer either by creating a new object to it or by calling the *getInstance* method, we are actually creating a thread of this class.



```
//create a socket
if( (sockfd = socket(AF_INET,SOCK_STREAM,0)) == -1 ){
    perror("socket");
    exit(1);
}

//Initialize the struct member for socket
their_addr.sin_family=AF_INET;
their_addr.sin_port = htons(PORT);

if (connect(sockfd,(struct sockaddr *)&their_addr,sizeof(their_addr)) < 0){
    perror("connect-socket");
    return -1;
}
//Initialize the struct member for message queue

mq_attr attr2;
attr2.mq_flags = 0;
attr2.mq_maxmsg = 10;
attr2.mq_msgsize = sizeof(myExecResult*);
this->mq = mq_open(mqName,O_CREAT | O_RDWR , 0, &attr2);



if (mq_unlink(mqName) == -1)
    return 1;

if (pthread_attr_init(&attr) != 0)
    return 1;

//This will create a thread for execTimer class
int retVal = pthread_create(&this->newThread,&attr,Run,NULL);
```

Figure 9: The content of *Startup* method

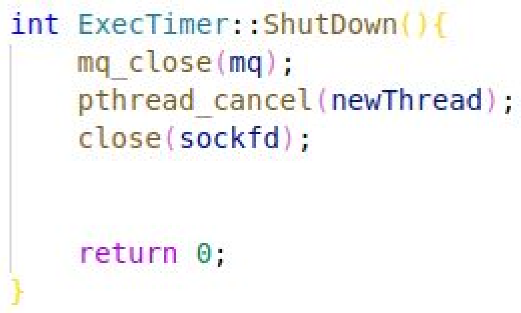
- **GetInstance:** We have created this function just to apply the singleton design pattern. Whenever we just need to call a function of this class, what do we need to do? We need to create a new object of this class. Assume that there are one thousand of these classes which use the functions of *ExecTimer* class. Does this mean we have to create one thousand new objects in each of these classes? No, our aim is to create only a pointer to the object of this class and initialize the pointer with the return value of GetInstance method. This prevents us from creating a lot of objects.
- **Begin:** This is the function that takes the beginning time of execution of the function. It is not returning anything, it basically assigns the beginning time to the *beginTime* field of the *ExecTimer*.
- **End:** All of the results of the execution time of the function is produced here. So, the function name is taken, and the difference time between the ending time of the execution and *beginTime* is calculated. Therefore, all of the data is in here. We need to send this data by using a message queue in this function.



```
void ExecTimer::End(const char * fn){

    myExecResult * results = new myExecResult();

    clock_t endT = clock();
    strcpy(results->funcName,fn);
    results->difference = (long)endT - beginTime;

    int res = mq_send(mq, reinterpret_cast<char*>(&results), sizeof(myExecResult*), 0);



}
```

Figure 10: The usage of mq_send to send a message via message queue

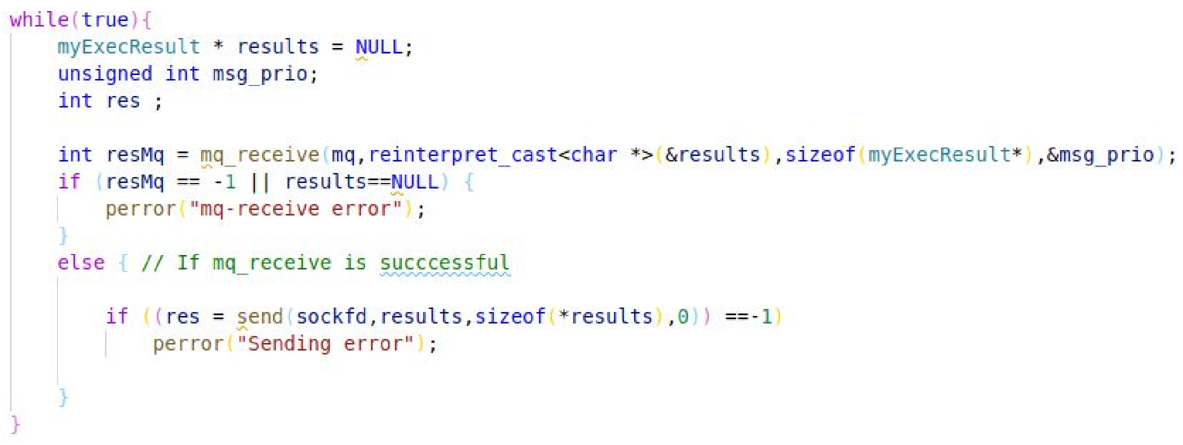
- **ShutDown:** This is the function which is called by the destructor of *ExecTimer*. We are closing the message queue and socket, and we are canceling the thread.



```
int ExecTimer::ShutDown(){
    mq_close(mq);
    pthread_cancel(newThread);
    close(sockfd);


    return 0;
}
```

Figure 11: Closing socket and message queue and canceling thread

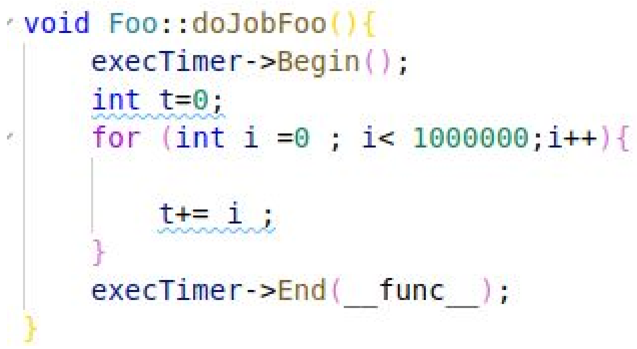
- **Run:** I can say that without this function, creating a new thread would be impossible, and surely this function is the most important function, whose name will be used while creating a new thread, in our class. We are receiving the message that is sent with the help of the message queue in the *End* method. Also, we are sending the results to the server side of our program by using the socket created in *StartUp*.



```
while(true){
    myExecResult * results = NULL;
    unsigned int msg_prio;
    int res ;

    int resMq = mq_receive(mq,reinterpret_cast<char *>(&results),sizeof(myExecResult*),&msg_prio);
    if (resMq == -1 || results==NULL) {
        perror("mq-receive error");
    }
    else { // If mq_receive is succcessful

        if ((res = send(sockfd,results,sizeof(*results),0)) ==-1)
            perror("Sending error");

    }
}
```

Figure 12: While loop in the *Run* method

The other classes have some functions in them, and they are using the ExecTimer's methods to measure their execution time. An illustration of how these functions should be used is shown below Figure 13.

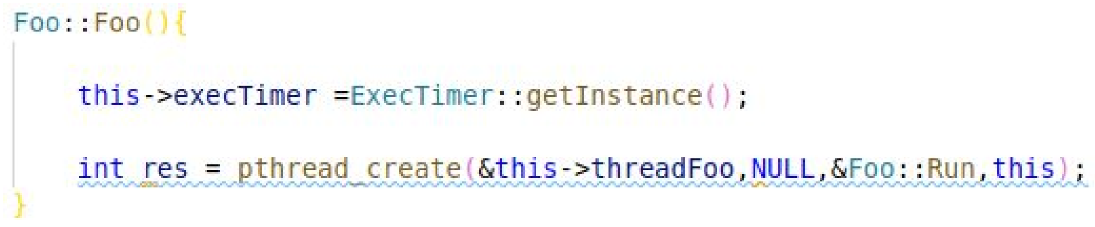


```
void Foo::doJobFoo(){
    execTimer->Begin();
    int t=0;
    for (int i =0 ; i< 1000000;i++){

        t+= i ;
    }
    execTimer->End(__func__);
}
```

Figure 13: An example usage of the functions of *ExecTimer*

As it can be seen in Figure 13, we are calling the methods with the help of *execTimer* pointer. This pointer is initialized in the constructor of the class by using the *getInstance* method of *ExecTimer*.

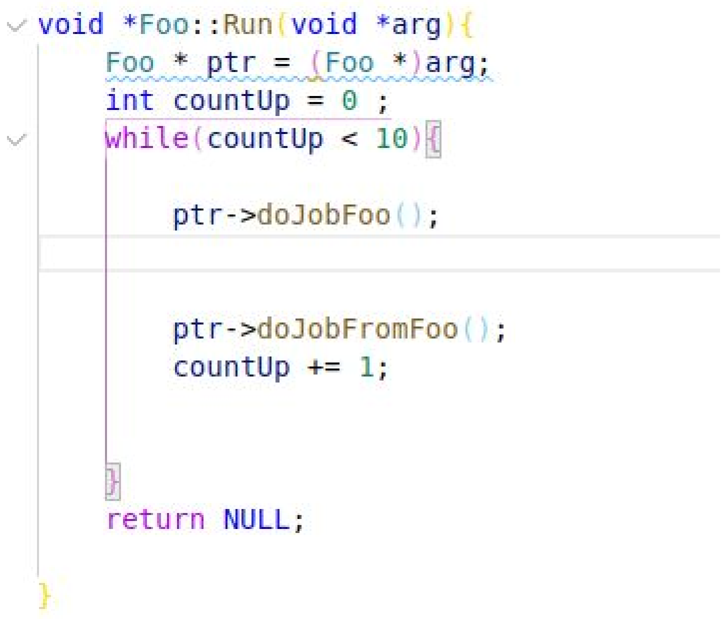
I also want to mention what if we are calling ExecTimer's methods in our thread. To achieve this, we need to create a thread of class in the constructor of the class. What we need is a pthread_t type variable and a static function which will call all of our functions in it. Here, the problem was that we are trying to access the methods of a class in a static function. The solution was creating a pointer to that class and initializing it with the casted value of the argument of *Run* function.



```
Foo::Foo(){

    this->execTimer =ExecTimer::getInstance();

    int res = pthread_create(&this->threadFoo,NULL,&Foo::Run,this);
}
```

Figure 14: Constructor of the class whose method's execution time is of interest

As it can be seen in Figure 14, in the constructor, we are initializing the *execTimer* pointer with the return value of *getInstance* method. Here, we are not creating a new object. We take the address of an object and assign it to our pointer. In addition, we are creating a thread of this Run method. The meaning of this is very important. When we are calling the constructor of a class, there will be a thread which is just created for the use of this class. Therefore, in an additional thread, the *Run* function will start to be working which is calling the methods of the function. Recognize that we are not creating an object of the class, or we are not calling the methods of the class by using an object to it. When the constructor is invoked, *Run* will be invoked. When Run is invoked, all of the methods of the class will be invoked.



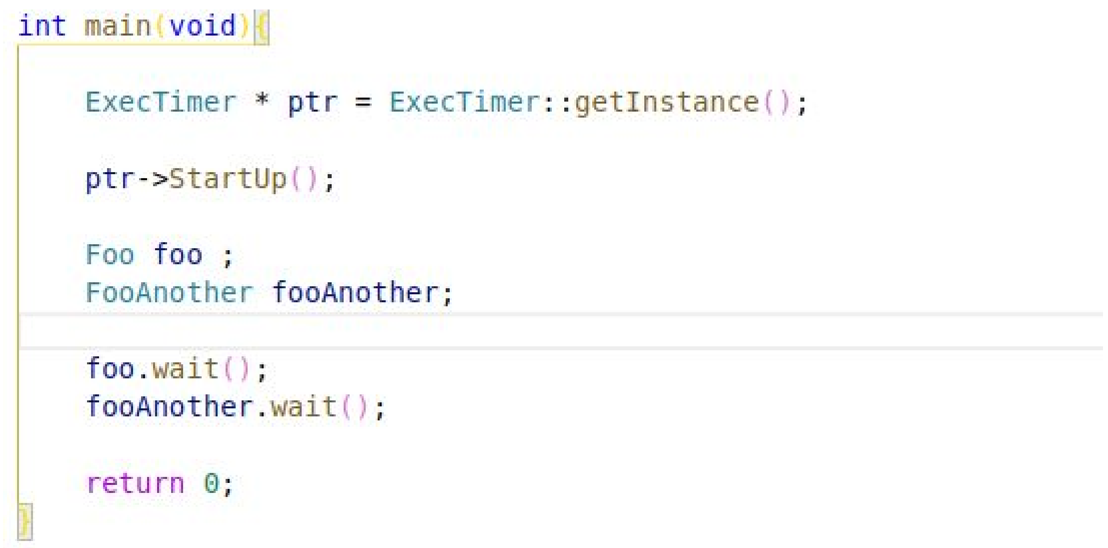
```
void *Foo::Run(void *arg){
    Foo * ptr = (Foo *)arg;
    int countUp = 0 ;
    while(countUp < 10){

        ptr->doJobFoo();


        ptr->doJobFromFoo();
        countUp += 1;


    }
    return NULL;

}
```

Figure 15: *Run* method of the class

In Figure 15, you will see the content of the *Run* method. Here, in the thread of the *Foo* class, we are calling the methods of the *Foo* class ten times. Now, we can have an idea about how multithreading can be handled. To make everything more generalized for other classes:

- Create the methods of the class
- In a static function that returns void pointer and takes void pointer as an argument, call all of your methods which belong to that class. This process can be done repeatedly for some number of times.
- In the constructor of the class, create a thread of this class by using the *Run* function of it.
- You are all done, whenever you are creating an object to this class, all the functions will be working within the thread of the class.

After you have finished the implementations of all of the methods, you will want to figure out the execution time of the methods in some runner program. Figure 16 shows how you can do this.



```
int main(void){

    ExecTimer * ptr = ExecTimer::getInstance();

    ptr->StartUp();

    Foo foo ;
    FooAnother fooAnother;

    foo.wait();
    fooAnother.wait();

    return 0;
}
```

Figure 16: How function calls are handled?

In Figure 16, after the call to the *startUp* function to *ExecTimer* class, a thread for the ExecTimer class will be created. After the creation of the Foo object, a thread for Foo will be created and our methods in the Foo class will be invoked within this thread. Finally, after fooAnother object is created, a thread for FooAnother will be created and our methods in the FooAnother class will be invoked within this thread. With all of these, we are creating a multithreaded application.

2.4 TESTING PHASE

Our project is a project where the aim is to measure the execution time of the functions from their threads. So, how can we test how much time is passed in order for a function to be executed?

There is no aim of this project such as testing since all of the input data of testing the program will be the functions whose execution time is of interest. So, to understand whether our program is performing correctly or not I created many functions that do some dummy jobs. These functions or methods should belong to the same class or different other classes. Here, the main focus is not to write these functions, but to see whether our data in the *ExecTimer* client sends the data correctly or not. Related to this topic, you can revisit Figure 5 and Figure 6.

3. ORGANIZATION

3.1 ORGANIZATION AND STRUCTURE

Advanced Technologies Research Institute (ILTAREN) is an institute operating under the TUBITAK Center of Research for Advanced Technologies of Informatics and Information Security (BILGEM), conducting research activities in Electronic Warfare (EW) .This research activities mainly focuses on field of modeling and simulation, spectrum management and also the development of embedded system prototypes as part of RF, IR optical systems. ILTAREN provides solutions in the form of projects, research and development programs, and also provides consulting support. ILTAREN is involved in work to fill the gap between the universities and the industry.

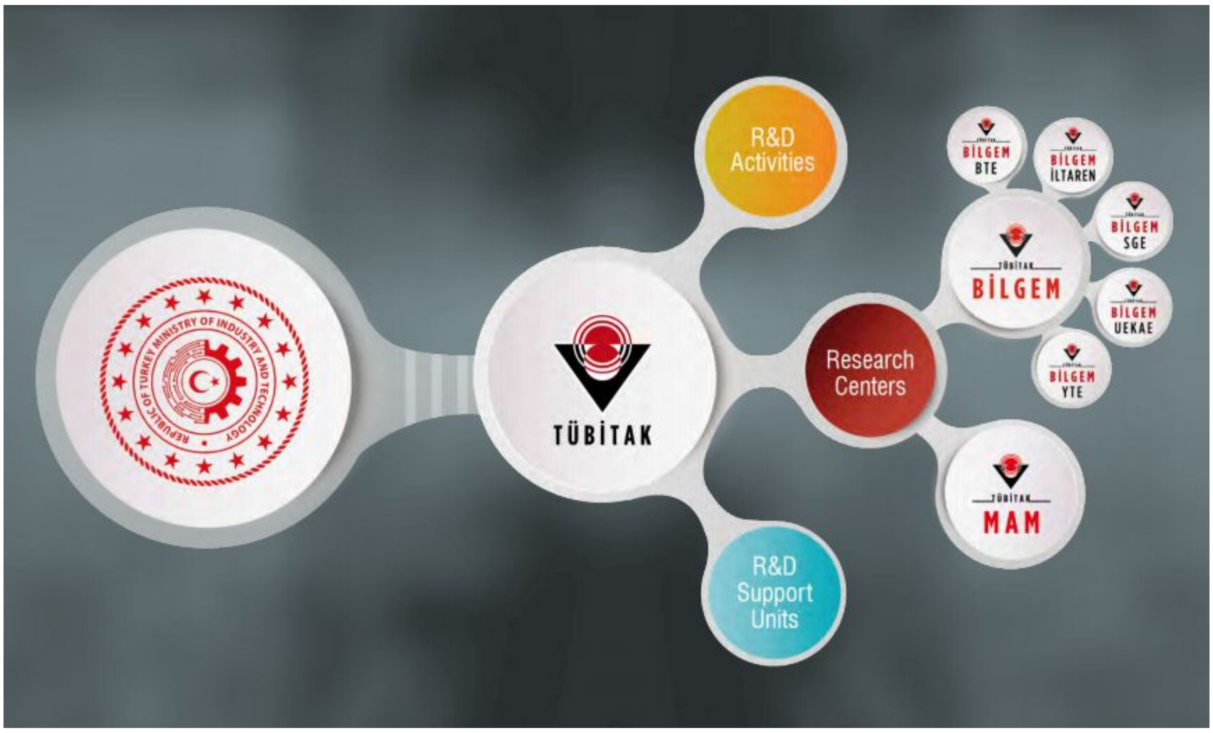


Figure 17: Tübitak Bilgem ILTAREN within Tübitak

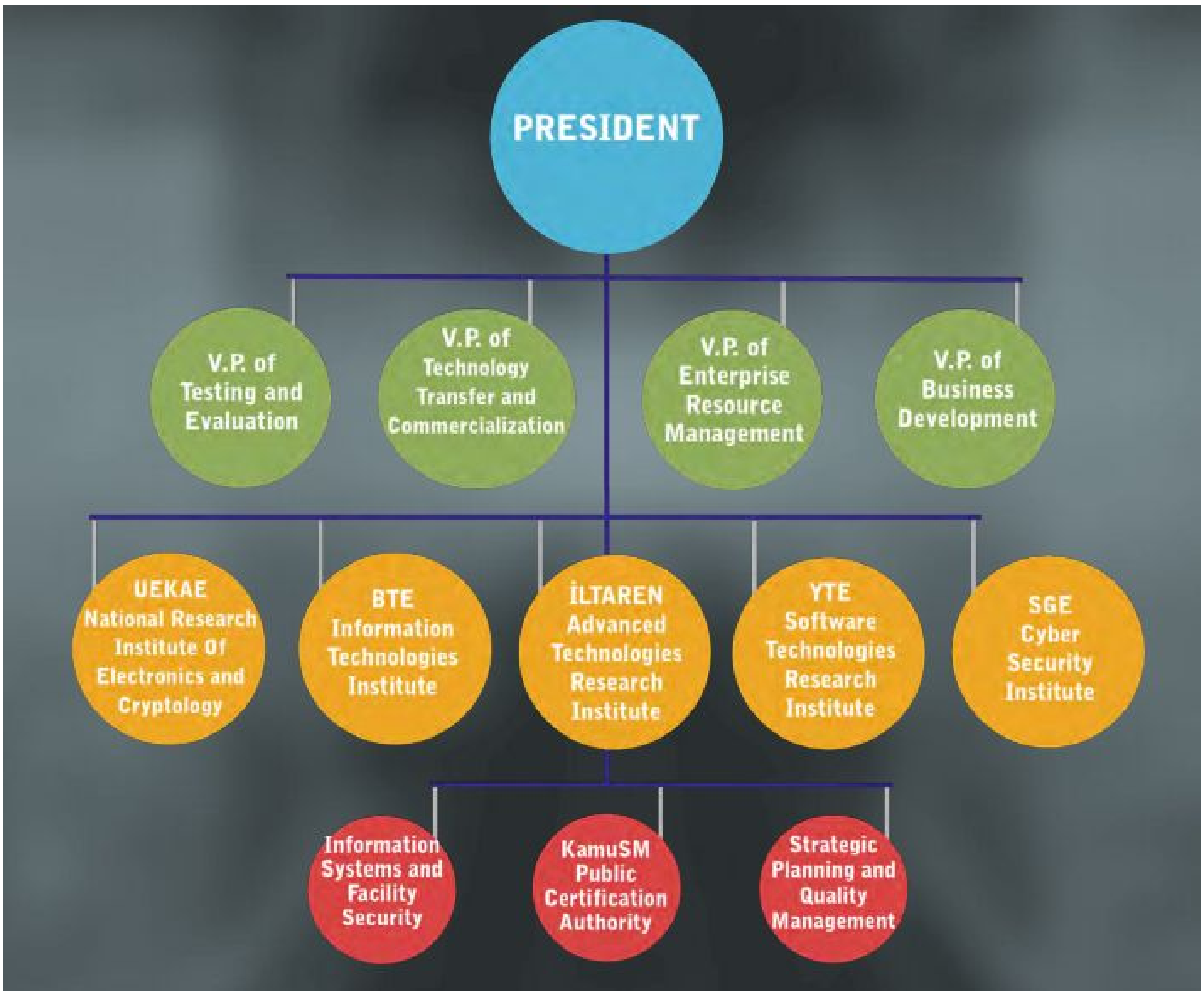

PRESIDENT
V.P. of Testing and Evaluation
V.P. of Technology Transfer and Commercialization
V.P. of Enterprise Resource Management
V.P. of Business Development
UEKAE National Research Institute Of Electronics and Cryptology
BTE Information Technologies Institute
İLTAREN Advanced Technologies Research Institute
YTE Software Technologies Research Institute
SGE Cyber Security Institute
Information Systems and Facility Security
KamuSM Public Certification Authority
Strategic Planning and Quality Management

Figure 18: Organizational structure

3.2 VISION AND GOALS

The vision of the organization is to become a base of science and technology that shapes the future by making our country a reference point in the fields of informatics, information security and advanced electronics.

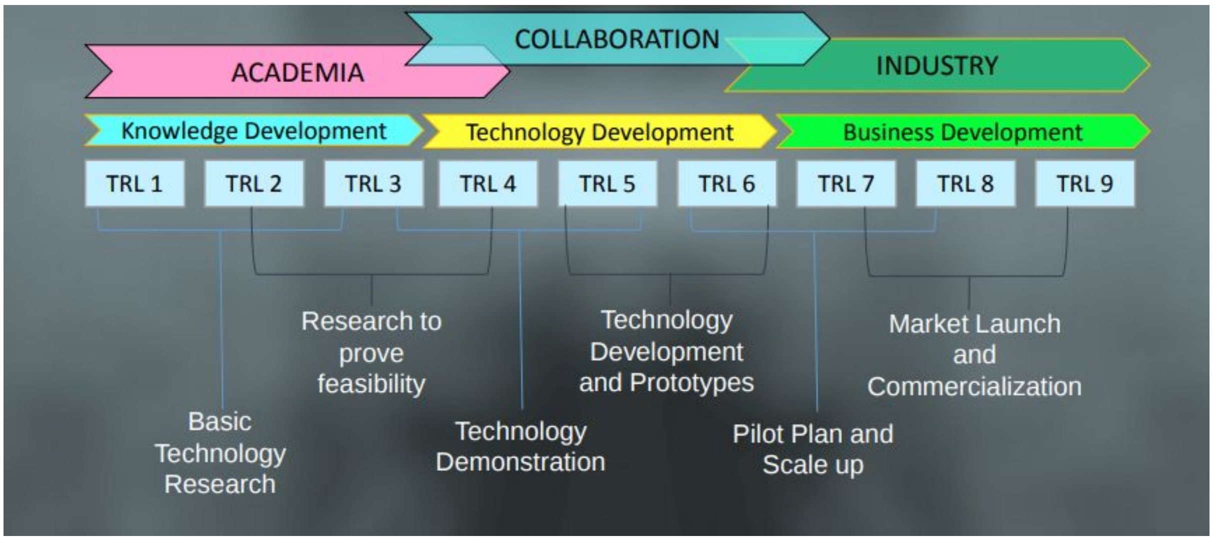


Figure 19: The mission of Tübitak Bilgem ILTAREN

The mission of the organization is to develop unique and high quality technologies that make the competitive power of our country permanent by making Ar-Ge studies, strategic collaborations with private sector and public institutions and the defense industry in the fields of informatics, information security and advanced electronics.

Carrying out studies in the fields of informatics, information security and advanced electronics that finds solutions to the needs of our company is one of the most important aims of the organization.

4. CONCLUSION

In conclusion, after having an internship at Tübitak Bilgem ILTAREN and creating the Execution Timer Application, I have learned the technologies, terms and concepts that I haven't known before. In addition to these, I had a chance to see what are the responsibilities of a backend software developer.

During my internship , the times that I have difficulties were :

- The time when I try to open a message queue
- The time when I try to create a thread
- The time when I try to use multithreading
- The time when I try to use built in library functions since I couldn't know what might be wrong about the execution

However, this was my first time having adventures with message queues, multithreading, network protocols. I felt that every information that I learned was something that I can use in my future career. Having some theoretical knowledge from the Internet and implementing these with a real life project was really satisfying.

In addition to these, during my internship, I recognized that the C++ knowledge that we have learned in our school, or the homeworks or projects that we conducted in the school is sufficient for us to directly dive into a C++ project and focus on some other concepts. In my project, I almost did not look at anything about C++ basics, or the concepts that we learn in our school. Having a very good background knowledge gained in my school, makes me focus on another concept that I don't know and understand the project in a very short amount of time.

5. REFERENCES

- About ILTAREN.(n.d.). Retrieved from: https://iltaren.bilgem.tubitak.gov.tr/en/kurumsal/iltaren

- Tübitak Bilgem ILTAREN within Tübitak.(n.d.). Retrieved from: https://bilgem.tubitak.gov.tr/sites/images/en-bilgem_corporate_presentation_v7-2019.04.09.pdf

- Organizational Structure.(n.d.). Retrieved from: https://bilgem.tubitak.gov.tr/sites/images/en-bilgem_corporate_presentation_v7-2019.04.09.pdf

- Mission of Tübitak Bilgem ILTAREN.(n.d.). Retrieved from: https://bilgem.tubitak.gov.tr/sites/images/en-bilgem_corporate_presentation_v7-2019.04.09.pdf

- Vision of Organization.(n.d). Retrieved from: https://bilgem.tubitak.gov.tr/tr/kurumsal/biz-kimiz