MQTT Alarm System

Group 7

Student ID	Name	Studies
au746215	Aleksander Chotecki	Computer Science
au746214	Natalia Potrykus	Computer Science
au672799	Marcin Szymanek	Software Technology
au705671	Tayib Enes Kalayci	Electronics Engineering

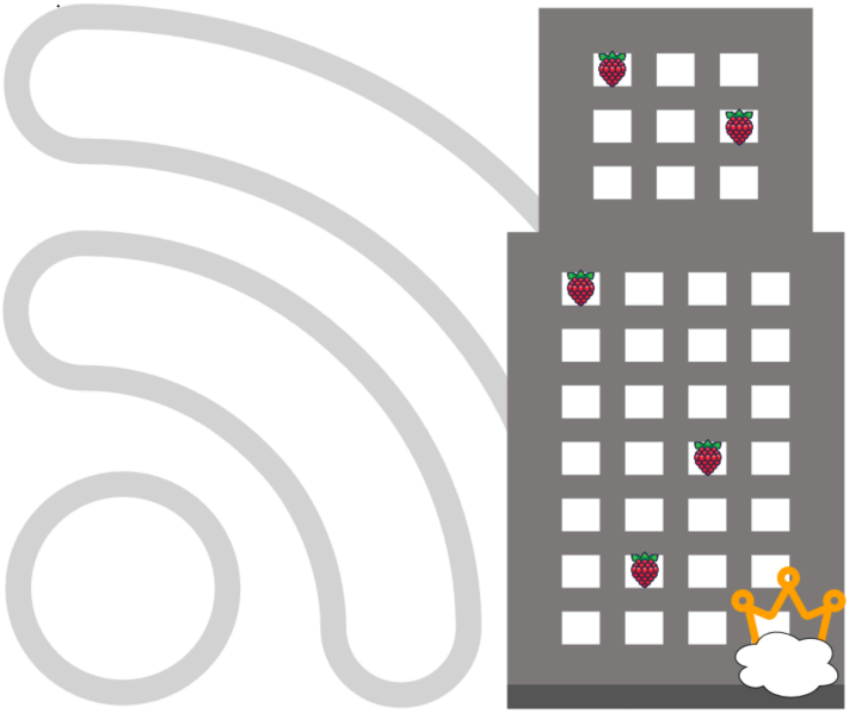


Table of contents

Work areas

Software - Aleksander Chotecki and Natalia Potrykus		
Work area	**Description**	**Primary responsible person**
System architecture	- Defining high-level structure of the components - Defining interactions between components - Creating diagrams	Aleksander Chotecki
Analysis of available technologies	- Researching available software libraries satisfying requirements regarding MQTT over TLS and safety	Aleksander Chotecki
PKI architecture design	- Designing authentication mechanisms to perform two-way authentication between clients and the broker - Creating diagrams	Aleksander Chotecki
Logic behind setting off alarms design	- Creating client-based alarm setting-off logic - Creating broker message distribution of alarm messages	Natalia Potrykus
Broker software	- Designing abstraction between software components	Aleksander Chotecki
GUI implementation	- Design and implementation of GUI with WPF for alarm information	Aleksander Chotecki
Client software	- Designing abstraction between software components - Implementing software	Natalia Potrykus

	components and prototyping sensors for hardware team	
PKI generator software	- Implementing logic of generating certificates (broker's and a certificate authority) - Implementing automatic distribution of certificates to broker and clients	Natalia Potrykus
Setting up TLS tunnel	- Distributing certificates to clients and the broker - Implementing tunnel between broker and clients	Natalia Potrykus
Creating dockerized client	- Mocking test data and creating docker images of clients to test the software	Aleksander Chotecki
Creating class diagrams	- Creating diagrams for client and broker software	Natalia Potrykus
Hardware		
Work area	**Description**	**Responsible person**
Setting up the prototype client devices	- Setting up the environment on the Raspberry PI - Adjusting client software to work on the Pi	Marcin Szymanek Tayib Enes Kalayci
Building the prototype clients	- Connecting the sensors w/ the client via ADC + spi interface - Setting up and controlling the buzzers	Marcin Szymanek Tayib Enes Kalayci
Miscellaneous	- Providing the components for the prototype - Video editing for the presentation	Marcin Szymanek

Introduction

Project Overview

The MQTT Smoke Alarm System is a project that demonstrates smoke detection through Raspberry Pi, sensors and subsequently triggers alarms as necessary. It uses Raspberry PI clients with a smoke detection sensor and a buzzer. Communication between broker and clients uses C# package - MQTTnet. Additionally, the project uses PKI for secure communication, and is compatible with .NET 7. Broker application includes a WPF-based GUI for broker management.

Why did we choose this project?

We decided to choose the MQTT Smoke Alarm System project due to its direct impact on people's safety. We wanted to try making a system that allows early alarms set off in case of an emergency. Moreover, we wanted to focus not only on safety, but also on security and cybersecurity as evidenced by the PKI implementation for secure communication - and such safety-critical software seemed to be a perfect use-case. We were able to put our cryptography knowledge in practice and experience some hands-on work with real hardware (raspberry PI). Also we were able to learn cooperation between teams with different pieces of knowledge in order to make a project.

Architecture and Design

The basic architecture uses Raspberry PI clients that communicate with the broker with MQTT protocol over Wi-Fi access point. The broker authenticates the client's device with username and password stored in *clients.json* file. The UI is written in C# WPF and it's coupled with the broker's software. The user is able to disable and enable the broker and see the clients' devices and their sensor's state with the UI.

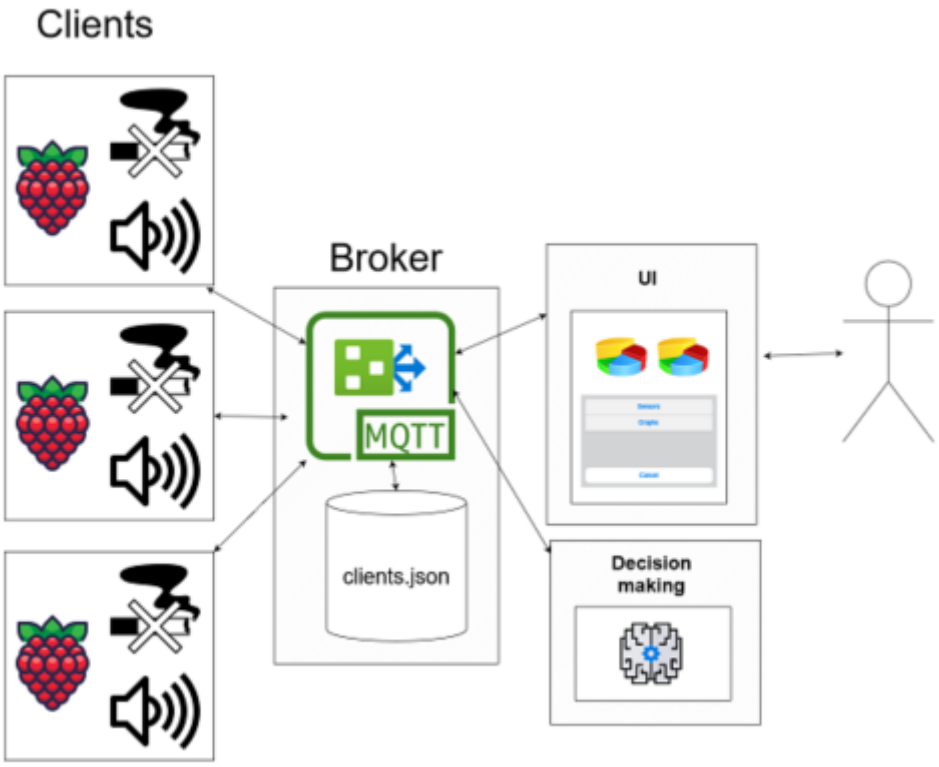


Figure 1 - basic architecture diagram

The client devices have sensors which let them detect smoke and in case of an emergency set off their buzzer. When the system is launched the clients connect and authenticate to the broker. In case of spotting a fire a device sets off their alarm and sends a message about their state being changed to the broker.

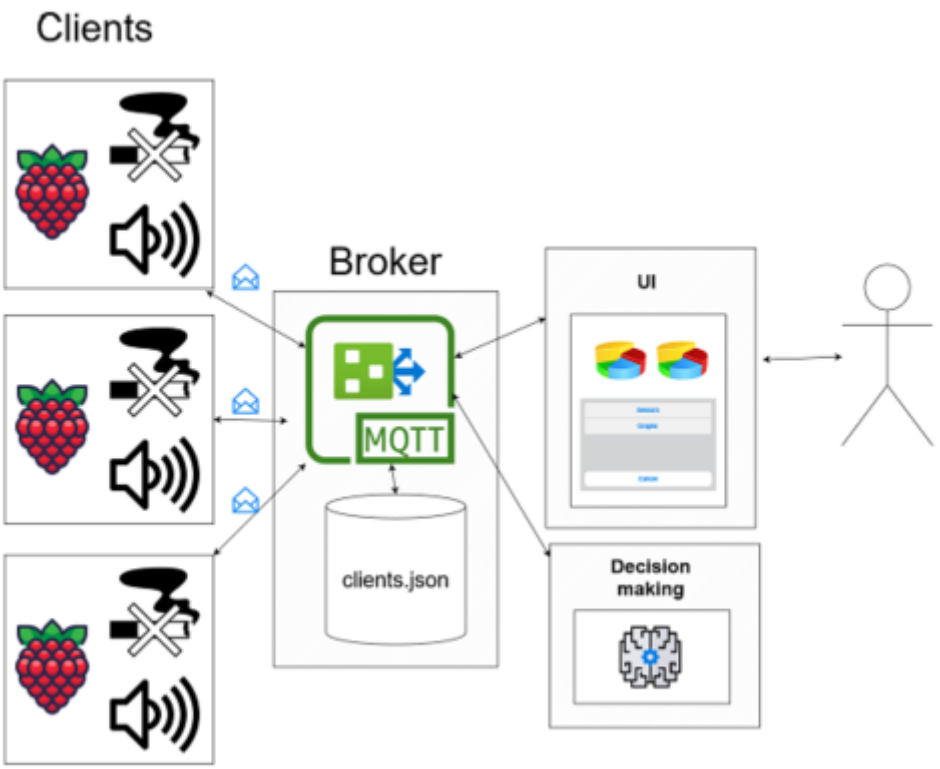


Figure 2 - Clients sending their state change information to the broker

In case of a client's state change, for instance detecting smoke, the client sends the message to the broker. The broker then broadcasts a message to the other clients to let them know they should enable their buzzers, because there is an emergency. The smoke detector state on the client's device is checked periodically, however updates to the broker are sent only in case of state change.

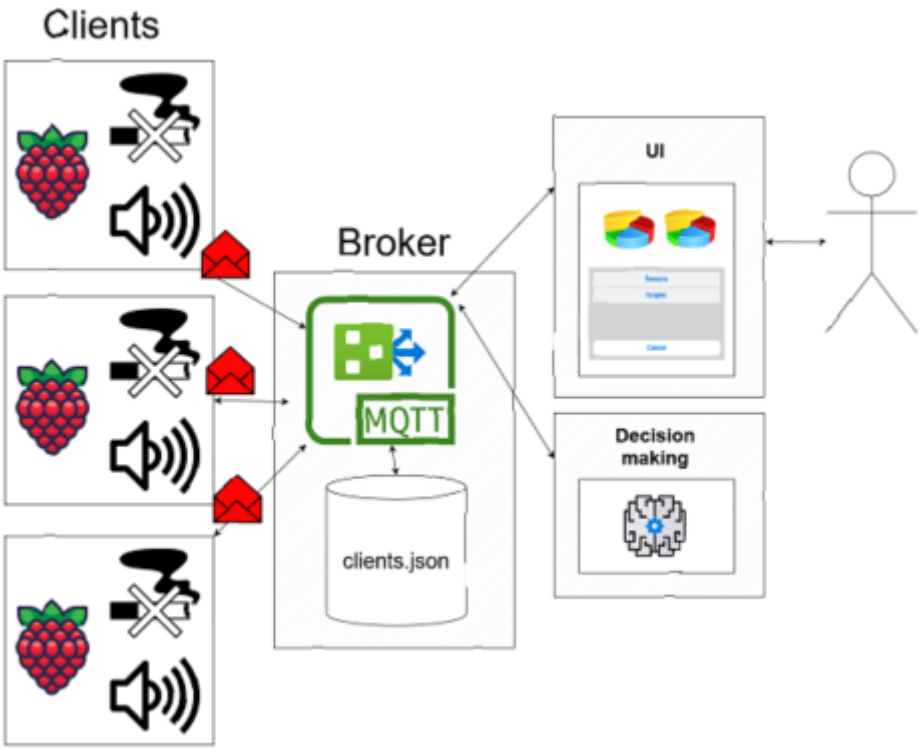


Figure 3 - The broker informs that there is fire and that other devices should enable their sensors.

To make the architecture more secure and not prone to man-in-the-middle attacks, clients establish TLS tunnels when connecting to the broker and they use MQTT over TLS so the messages about the smoke detection remain secure and unmodified.

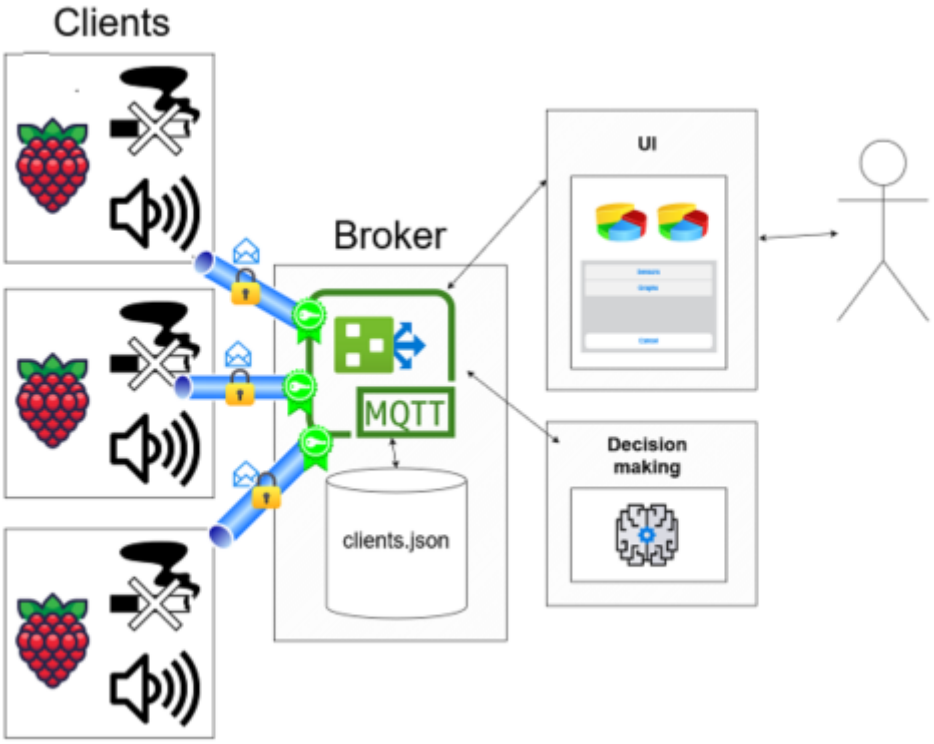


Figure 4 - TLS tunnels with certificates served by the broker

Background and/or Technical Analysis

Technological choices - software

Possibilities

The technological choices for the software part included such aspects as:

- **possibility of local deployment of the broker -** we wanted the system to be resistant to network failures as much as it is possible. That's why we wanted to have control over broker deployment and where it is placed - whether it's a local broker inside the protected building or further from the local network. We decided to insist on flexibility and setting our own constraints of the system, therefore we did not go for any of the cloud providers.

- **possibility of implementing TLS** - we read the libraries documentation carefully whether the software allows to implement our own public key infrastructure

- **possibility of extensive, flexible configuration capabilities** - we wanted to have control over the configuration and the code - that is why we decided to go with MQTTnet library, which has an API that allows for a great adaptability of broker's and communication settings

- **technological familiarity** - we wanted to use programming languages that everybody feels comfortable with

- **possibility of manipulating sensors placed on Raspberry Pi with ease** - we checked for availability amongst popular languages. Both Python and C# offered external packages to handle RPI communication.

Final technology choices

Technology	**Why?**
Locally-run broker	Possibility to build and manage Public Key Infrastructure (certificates, username-password authentication)
MQTTnet C# library	Extensive control over broker configuration, programming language familiarity
Pythonnet library	Out-of-the-box possibility to integrate python code necessary to manage the hardware into our C# solution. Turned out to be crucial, due to problems with getting RPI to work with C#
WPF	(Seemingly) easy option for UI application of C#-based option.

We had plenty of possibilities for MQTT software libraries or standalone applications that meet our requirements. The most popular ones include for example Mosquitto, which is a standalone broker application. There is of course a possibility to pair it with for instance the Paho-client Python library for client-side functionality or any other client-side software. Another option that has been considered is HiveMQ which is a broker that we could possibly deploy as a separate application with an external client. Finally, there's MQTTnet, a library in C# that provides both client and broker functionalities, allowing manual coding and a lot of control.

Having considered our team's familiarity with C# language and our preference for extensive control over configurations using code we opted for the MQTTnet package. This choice allowed us to implement both the client and server sides of our MQTT system using the same technology (language and packages) and manipulate the configuration in a flexible way and according to our specific needs.

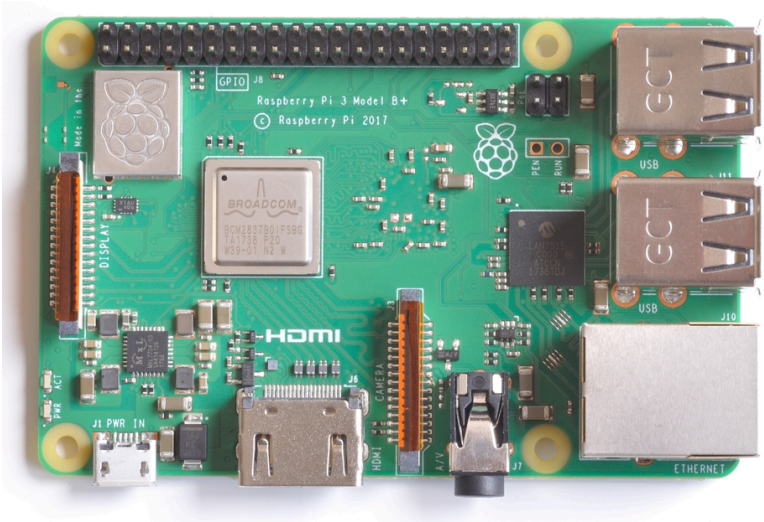
Technological choices - hardware prototype

Possibilities

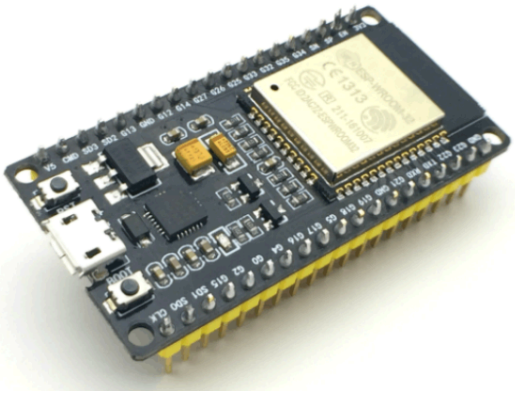
SoC:

The key factors in choosing the necessary system on chip were availability, support for .NET and development speed. For the client we were considering a couple of choices:

- **RaspberryPi 3b+ -** This small computer is readily available from vendors in Europe, comes with a Linux based OS, a lot of GPIO's, an SPI interface and more than enough RAM and processor speed. It has a couple of disadvantages: running Linux means it is not necessarily the best choice for time-sensitive tasks, since the OS takes care of a lot of scheduling work. It also does not come with any analog pins or ADC out of the boot, so an external ADC is required. It's GPIO's, however, are more than capable of driving a small buzzer, provided it is a 3V buzzer.



- **ESP32 -** ESP32 is a popular microcontroller, sporting Wifi/Bluetooth functionality, low cost and low power requirements. Since it is a popular choice among many IoT projects both small and large scale, it would be a mistake not to consider it.



- **RaspberryPiZeroW** - this is a headless model of a RaspberryPi with lower cost and specs. Reading from the sensor would be even easier than with RaspberryPI3b+, since we have previously used ASE-fHAT, a Hardware Attachment on Top, which includes an ADC, making it possible to read analog input voltage directly from the sensor.

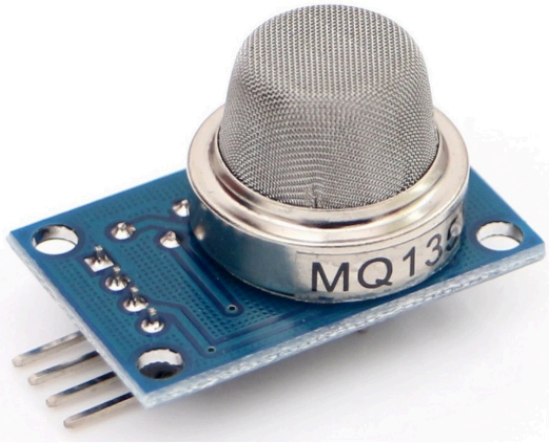
Final SoC choice:

We have decided to go with **RaspberryPi 3b+.** This is because a lot of the work on the client was done in advance using dotnet, and the other SoC's with inbuilt Wifi capabilities do not have an architecture supported by dotnet, so implementing the client in those would require extensive rewriting of the client code using solutions which are not MQTTnet. On top of that, the team has a lot of experience using RaspbianOS and setting up the Pi, so it was a straightforward choice.

Smoke Sensors:

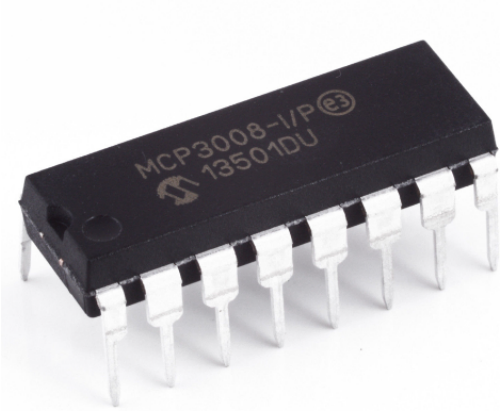
For the smoke sensors, the biggest factor was price: we wanted to use something that could be bought cheaply so that we could build the prototype with minimum

costs. For that reason we chose the MQ-135 **Air Quality Sensor.** Most smoke particles contain an ample amount of CO2 and MQ-135 is perfectly capable of detecting those. Since **MQ-135** can be bought for under 8$, we deemed it suitable for the project.



ADC:

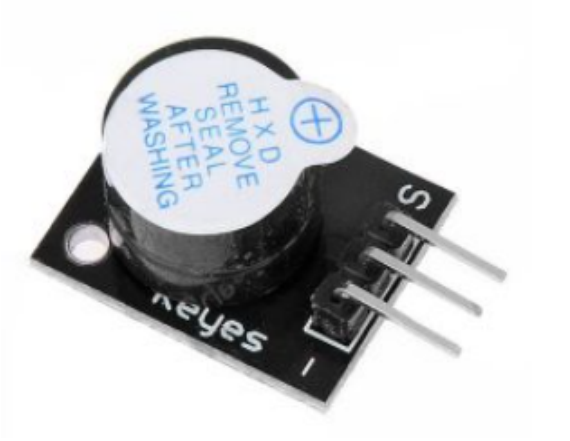
Since RaspberryPi do not come with analog input pins, we had to use an ADC. Our choice here was the MCP3008: an ADC which can take up to 5V input, has an SPI interface and can use 3.3V digital signal for the SPI interface. Any ADC which does not support 3.3V digital output would need some kind of a circuit to convert its output to 3.3V, and since the group does not have extensive experience in electronic circuits, that would have been a poor solution.



Buzzer:

The only concern for the buzzer was that it could be driven by a 3.3V GPIO pin from the Raspberry Pi, and that the cost would be as low as possible. We have opted for

KY-012 Active Buzzer 3.5-5.5V, since that was one of the few buzzers which could be driven by less than 5V out of the box.



The datasheets for all of the components can be found at the project git repository .

Additional Hardware considerations

The prototype which we built is just that - a prototype. There are many factors to consider, to actually turn this into a real product.

- Reduce the power draw of the SoC. For a system which is not supposed to do more than gather data, control a buzzer and send data out, there is no need to use a RaspberryPI 3B+. For a straightforward comparison, RasberryPi pico when running an example program draws an average of 85-95 mA source (chapter3), while an idle current consumption of RaspberryPi3b+ is around 300mA source.

- Replace 3.5 V buzzer with a stronger device. It would be prudent to choose a buzzer which can produce a loud sound, for example in the 90-120 dB range. This would require a circuit and a power supply that could support at least 9-12 V. Devices such as this one promise 120dB while using 12V.

- Do more extensive research on the smoke sensors and consider using several sensors at the same time. Currently we only use a single MQ-135 air quality sensor per device, but it might be a good idea to compare different sensors to each other and maybe even use several sensors on each device. This would make sure that if one of them fails, at least the other ones are still working.

- Add a temperature sensor. If the point is to alarm the user in case of a fire, the temperature sensor seems like a clear upgrade to the system. Additionally, it would be a great data source to send to the broker, so that any interested systems could collect that.

- Extra battery in case of failure. Currently, when power from the RaspberryPi power supply fails, the whole system fails. An alarm system might benefit from an extra battery to make sure that the system works for as long as possible.

Why is it an IOT project?

The system is an IOT project because:

- **It has a broker that passes messages around clients** - the system controls real-time communication between client devices that subscribe to the broker

- **The client devices also use real hardware** - the devices collect real-time data about the smoke which is published on the broker. They use real, physical smoke detector sensors and are able to buzz when receiving a proper message

- **Clients connect to the broker with MQTT protocol** - clients use a pub/sub communication model which is typical for IOT devices.

- **There is real time data collection -** the client devices collect periodic data about the environment detected by the sensors

- **The solution is scalable to a sufficient extent** - the solution should be able to handle dozens of devices located in a building. We tested it with several docker-simulated clients and it worked without any problems. The MQTTnet documentation does not explicitly say what is the maximum capacity of the solution, however MQTTnet github repository suggests it to be in thousands of clients with hundreds or thousands messages per second. So we consider it sufficient for our requirements.

Implementation

Software

Software part in our project consists of 3 separate C# projects serving different purposes.

Client (Raspberry Pi Client)

The Client project represents the Raspberry Pi client software. It is responsible for collecting sensor data, detecting smoke, and communicating with the broker using MQTT over TLS for secure data transfer. Clients can be deployed both on Windows and Linux.

Used packages:

- *.NET 7*
- *MQTTnet 4.3.1.873*
- *MQTTnet.Extensions.ManagedClient 4.3.1.873*

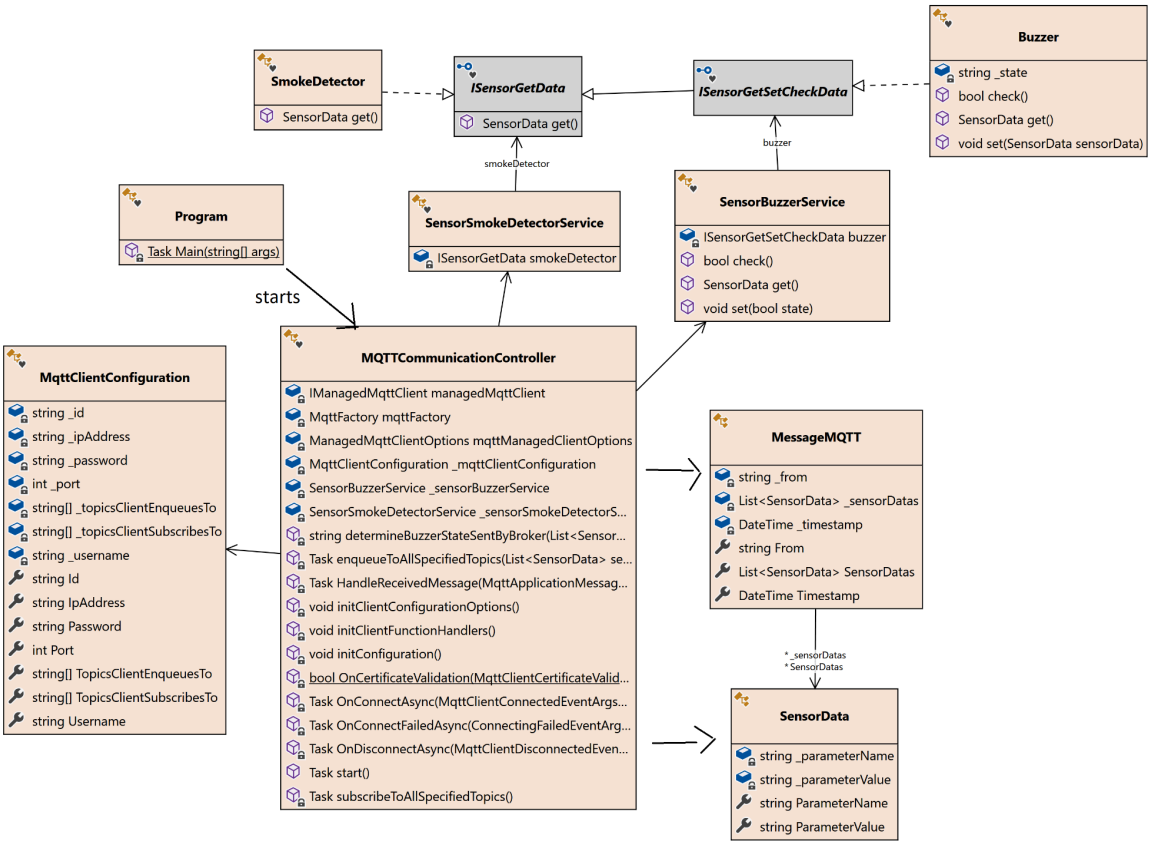


Figure 5 - Client class diagram

Broker

The Broker project serves as the central communication hub that manages multiple clients and their respective sensors. Additionally, there is a graphical interface developed with WPF for interacting with the broker. It monitors clients' sensor state and displays alarm state. Use of WPF makes it work only on Windows.

Used packages:

- *.NET 7*
- *MQTTnet 4.3.1.873*

- *Newtonsoft.Json 13.0.3*

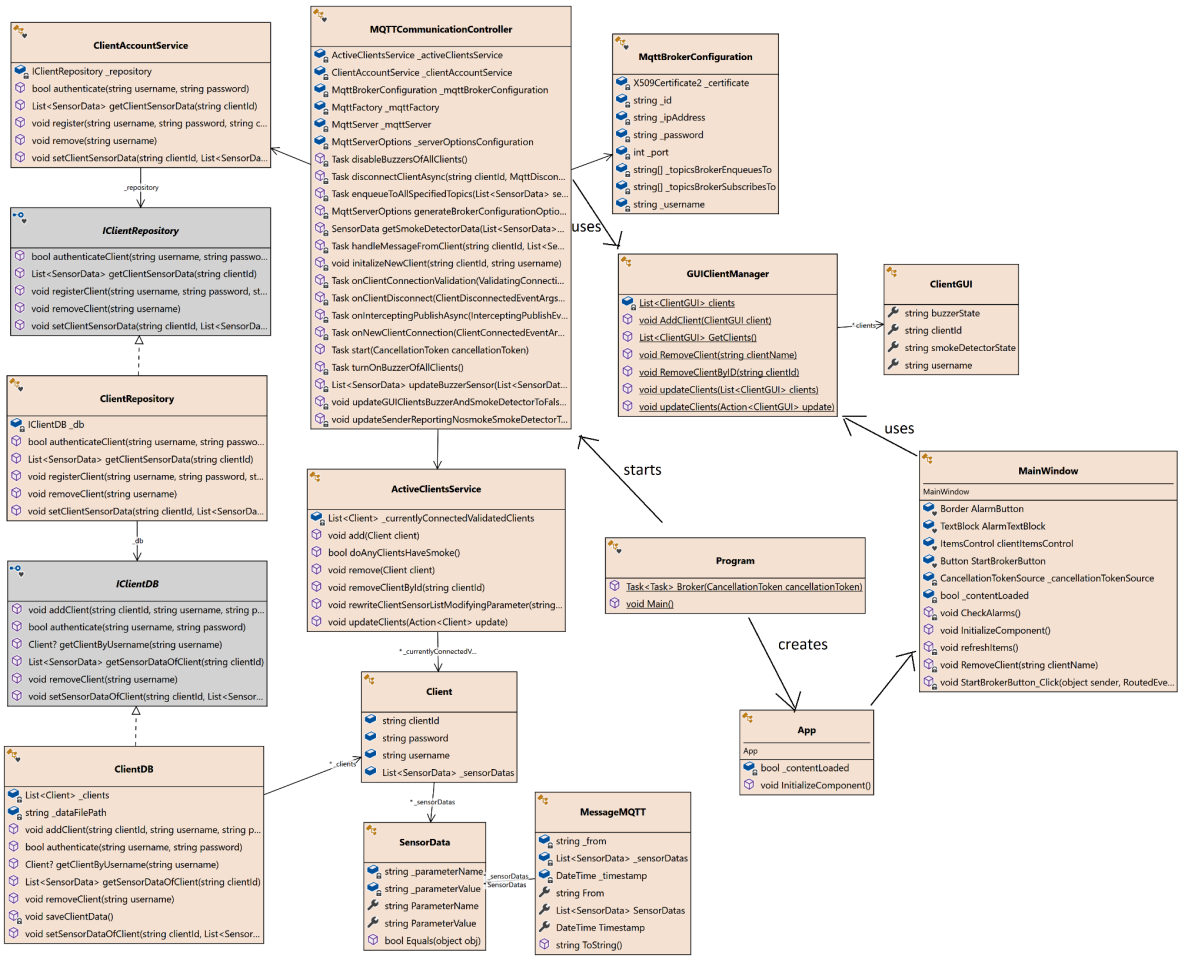


Figure 6 - Broker class diagram

Figure 7 - Broker GUI made with WPF - two views: no alarm (left), alarm (right)

PKIGenerator (Public Key Infrastructure Generator)

The PKIGenerator project is responsible for generating the necessary Public Key Infrastructure (PKI) for both clients and the broker and saving keys and certificates in relevant directories of the broker and clients

Software logic

When it comes to detection smoke and enabling sensors the system uses the following logic:
Whenever a change of smoke state is detected on any client device:

If there is smoke detected by one client device:

1. Broker passes the message {BUZZER:TRUE} to all authenticated clients and updates local authenticated clients list {BUZZER:TRUE}

If one client device stops detecting smoke:

1. Broker updates client's smoke sensor data in the list {SMOKE:FALSE}
2. If smoke status is FALSE everywhere (there is no smoke detected anywhere anymore): enqueue message {BUZZER:FALSE} on the channel for all clients

For every step, there is a proper GUI update called.

Relevant code:

```
if (smokeSensorData.ParameterValue == "TRUE")
{
      await turnOnBuzzerOfAllClients();
      updateGUIAllClientsBuzzerTrueAndSmokeDetectorOfSenderTrue(cli
      entId);

}
else
{
      updateSenderReportingNosmokeSmokeDetectorToFalse(clientId);

      // if smoke detector is down everywhere (if fire didnt spread)
      if (!_activeClientsService.doAnyClientsHaveSmoke())
      {
      await disableBuzzersOfAllClients();
      updateGUIClientsBuzzerAndSmokeDetectorToFalse();

      }
      else
      {

updateGUIClientReportingNosmokeSmokeDetectorToFalse(clientId);

      }
}
```

What if clients lose connection with the broker?

Clients also have logic responsible for enabling buzzers locally if they cannot connect to the broker, which means that if there is fire there will be a buzzer enabled in the room that fire starts.

Relevant code:

```
  if (managedMqttClient.IsConnected)
  {
if(!(ParameterValueSmokeDetectedNew ==
ParameterValueSmokeDetectedOld))
     {
     await enqueueToAllSpecifiedTopics(sensorDatas);
     }
     else
     {
          Console.WriteLine(“state NOT CHANGED”);
  }
  else
  {

     if(!ParameterValueSmokeDetectedOld &&
ParameterValueSmokeDetectedNew)
     {
          _sensorBuzzerService.set(true);
          Console.WriteLine("BUZZER ENABLED BY LOCAL SYSTEM");
     }
     else if(ParameterValueSmokeDetectedOld &&
!ParameterValueSmokeDetectedNew)
     {
          _sensorBuzzerService.set(false);
          Console.WriteLine("BUZZER DISABLED BY LOCAL SYSTEM");
     }

  }
```

Security - how is authentication implemented?

The public key infrastructure has one certificate authority that is distributed to all of the clients. Broker's certificate is signed by the certificate authority and the broker uses both the certificate and the key that corresponds to the public key in this certificate.

All of the clients are connected with TLS tunnels (MQTT over TLS). There is mutual authentication between clients and the broker, because during connection the client gets the broker's certificate and checks the trust chain against the certificate authority (previously generated and supplied to the client). On the other hand, the broker is able to verify clients that supply him with username and password that are checked in the broker's local clients.json file. Therefore, Clients provide credentials through a secure tunnel, because it's happening after TLS negotiation and there is mutual authentication of parties.

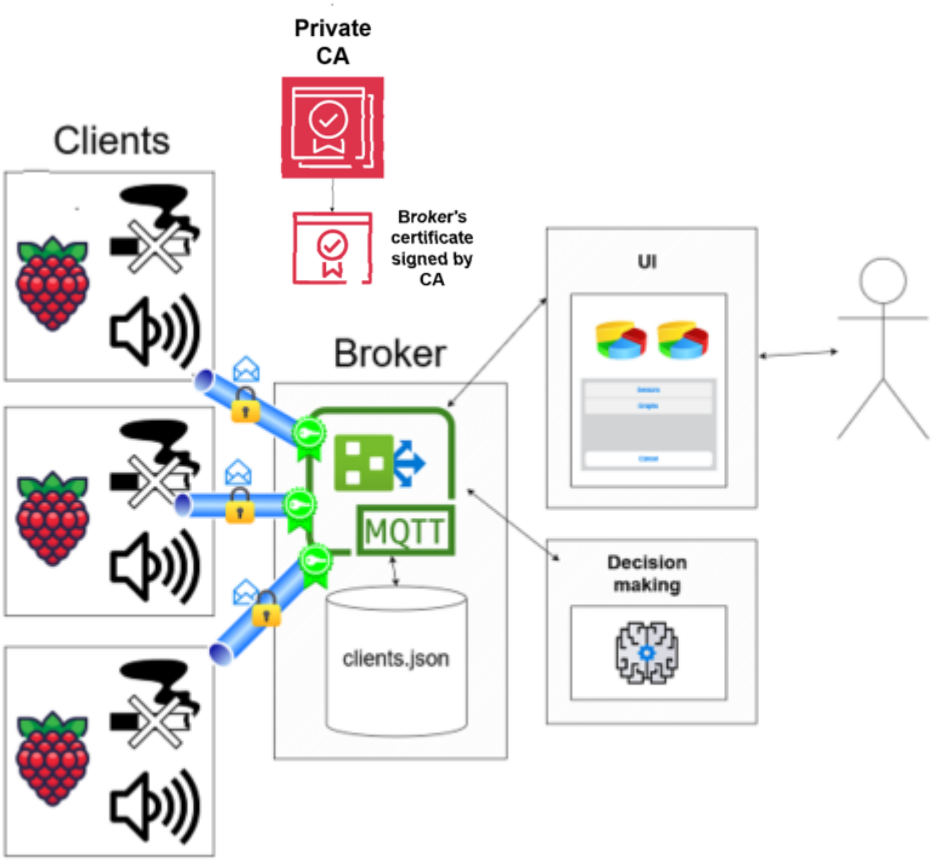


Figure 8 - Secure architecture and PKI (CA + broker's certificate)

Relevant code:

1. Generating broker's certificate with chain and CA digital signature

```
static X509Certificate2 GenerateSignedCertificate(string subjectName, string keyFilePath, X509Certificate2 issuerCertificate, string password)
{
	using (RSA rsa = RSA.Create(2048))
	{
	var request = new CertificateRequest(new X500DistinguishedName($"CN={subjectName}"), rsa, HashAlgorithmName.SHA512, RSASignaturePadding.Pkcs1);
	var certificate = request.Create(issuerCertificate, DateTimeOffset.Now, DateTimeOffset.Now.AddYears(8), Guid.NewGuid().ToByteArray());

	X509Chain chain = new X509Chain { ChainPolicy = { RevocationMode = X509RevocationMode.NoCheck, VerificationFlags = X509VerificationFlags.AllowUnknownCertificateAuthority } };
	chain.ChainPolicy.ExtraStore.Add(issuerCertificate);

	if (chain.Build(certificate))
	{
		var pfxBytes = certificate.Export(X509ContentType.Pkcs12, password);
		var pfxCertificate = new X509Certificate2(pfxBytes, password, X509KeyStorageFlags.Exportable | X509KeyStorageFlags.PersistKeySet);
		ExportPrivateKeyToPemFile(rsa, password, keyFilePath);
		return pfxCertificate;
	}
	else
	{
		throw new Exception("Certificate chain validation failed.");
	}
	}
}
```

2. Reading broker's certificate and adding it to the configuration options

```
private MqttServerOptions generateBrokerConfigurationOptions()
{
	return new MqttServerOptionsBuilder()
.WithEncryptedEndpointPort(_mqttBrokerConfiguration.Port)
.WithEncryptedEndpointBoundIPAddress(IPAddress.Parse(_mqttBrokerConfiguration.IpAddress))
.WithEncryptionCertificate(_mqttBrokerConfiguration.Certificate.Export(X509ContentTy
```

```
pe.Pfx))
.WithEncryptionSslProtocol(System.Security.Authentication.SslProtocols.Tls12)
.WithEncryptedEndpoint()
       .Build();
}
```

3. Reading broker's certificate and adding it to the configuration options

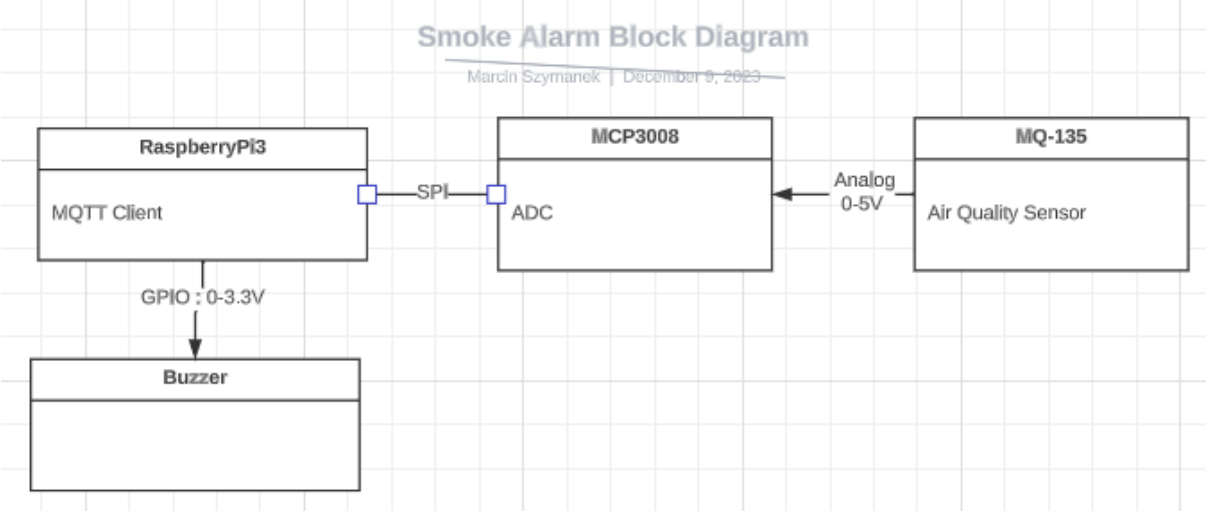
```
private static bool OnCertificateValidation
(MqttClientCertificateValidationEventArgs args)
{
       string    path    =    Environment.GetEnvironmentVariable("CA_PATH")    ??
"../../../PKI/CA/rootCA.cer";

       X509Certificate2 serverCertificate = new X509Certificate2(args.Certificate);
       X509Certificate2                        CACertificate                          =
PKIUtilityStatic.ReadCertificateFromFile(path);

       try
       {
       args.Chain.ChainPolicy.TrustMode = X509ChainTrustMode.CustomRootTrust;
       args.Chain.ChainPolicy.CustomTrustStore.Add(CACertificate);
       args.Chain.ChainPolicy.RevocationMode = X509RevocationMode.NoCheck;

       var chain = args.Chain.Build(serverCertificate);

       args.Chain.ChainStatus.ToList().ForEach(x                                    =>
Console.WriteLine(x.Status.ToString()));
       Console.WriteLine(chain  ?  "OnCertificateValidation:  Building  certificate
chain success" : "OnCertificateValidation: Building certificate chain FAILURE");

       return chain;
       }
       catch (Exception ex)
       {
       Console.WriteLine(ex);
       Console.WriteLine(ex.StackTrace);
       return false;
       }
}
```

Hardware

The hardware part of the implementation is relatively simple, since the only requirements were:

- Read from the smoke sensor
- Control the buzzer

The following is a block diagram representing the system and its components.



Both MCP3008 and MQ-135 are powered by the RaspberryPi with 3.3V and 5V DC respectively.

Since we borrowed one of the smoke sensors, it wouldn't make much sense to make a PCB of the prototype, so we decided to stick to a breadboard prototype. It's worth noting that MCP3008 could potentially support up to 8 separate channels, so up to 8 analog sensors could be used, however there is not guaranteed that RaspberryPi would be able to supply all of them with power.

Reading from the sensor

Initially we were planning to use dotnet's Iot libraries to read from the ADC, since there is even an MCP3008 class specifically for our needs. Sadly we were not able to use these libraries after getting an error while trying the example code from this tutorial.

```
/home/pi-smoke0/IoT/MQTTSmokeAlarmSystem/Client/Sensors/SmokeDetectorMcp3008.cs(15,
29): error CS0144: Cannot create an instance of the abstract type or interface
'SpiDevice' [/home/pi-smoke0/IoT/MQTTSmokeAlarmSystem/Client/Client.csproj] 34
Warning(s) 1 Error(s)
```

This indicates that the class used in the code is Abstract and not implemented. We believe this might be a version problem, where the class has changed somewhere

during the development of the library, but decided not to spend time investigating the issue. Instead we opted for using python code embedded in C# using pythonnet library, using the gpiozero python library. The library was trivial to test and worked out of the box, also with a MCP3008 class, so the only issue was: how do we call the python code from our client? Pythonnet allowed us to do exactly that: call python code from C#:

```
 public SmokeDetectorMcp3008(){
            threshold =
float.Parse(Environment.GetEnvironmentVariable("BUZZER_THRESHOLD"));
            Console.WriteLine("Alarm state threshold: " + threshold.ToString());
            PythonEngine.Initialize();
            using(Py.GIL()){
                dynamic gpiozero = Py.Import("gpiozero");
                mcp = gpiozero.MCP3008();
            }
        }

        public SensorData get()
        {
            float val;
            using(Py.GIL()){
                val = mcp.value;
            }

            Console.WriteLine("Mcp3008 data:");
            Console.WriteLine(val.ToString());

            string isThereSmoke = "FALSE";
            if(val > threshold){
                isThereSmoke = "TRUE";
            }

            return new SensorData("SMOKE", isThereSmoke);

        }
```

Note the call to parse the environment variable: after testing we found out that the different sensors have different response to similar concentration of smoke. This is likely because different resistors were used for the different batches of the smoke sensors. We initially thought that the resistance could be changed via a potentiometer on the sensor, but quickly found out that that was not the case. Therefore, we set the smoke threshold for each sensor individually via environment variables.

The mcp.value call returns the result of the ADC in float format: a value between 0 and 1 corresponding to the input voltage from the sensor between 0 and 5 Volts. This is then compared to a threshold value to figure out if the state of the alarm should be changed.

It's worth mentioning that pythonnet was not technically needed, another solution could be to continuously read in a separate process and store results from a file, and then read from that file in our C# client, and surely there are other solutions which we did not think of, that could solve this problem as well.

Controlling the buzzer

We stumbled upon a few problems with the buzzer. First off, the buzzer did not work as expected. It has 3 pins: Signal, V+ and Ground. When 5V would be applied to the signal pin, the buzzer made a sound. This was not the same with 3.3V used by RaspberryPi GPIO.
According to the datasheet it should be possible to turn on the buzzer by applying 3.3V to the V+ pin and 3.3V to the signal pin. This was not the case. In fact, it seemed that the middle pin was not connected to anything at all.

After extensive research, we found out that applying a square wave in 2-4kHz frequency would produce the desired buzzing sound. This created another problem: C# does provide the ability to sleep for less than a millisecond, and according to this post on stackoverflow the Thread.Sleep() method actually blocks for 12-15 ms. To circumvent this we created a simple python script that will buzz continuously in a more or less desired frequency and launched that as a separate process.

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
pin = 24
GPIO.setup(pin, GPIO.OUT, initial= GPIO.LOW)

freq = 1000*2

print("Buzzer-test")

try:
  while True:
    GPIO.output(pin, GPIO.HIGH)
    time.sleep(1/freq)
    GPIO.output(pin, GPIO.LOW)
    time.sleep(1/freq)

except KeyboardInterrupt:
```

```
  GPIO.cleanup()
```

When the buzzer should be turned off, we simply kill the process.

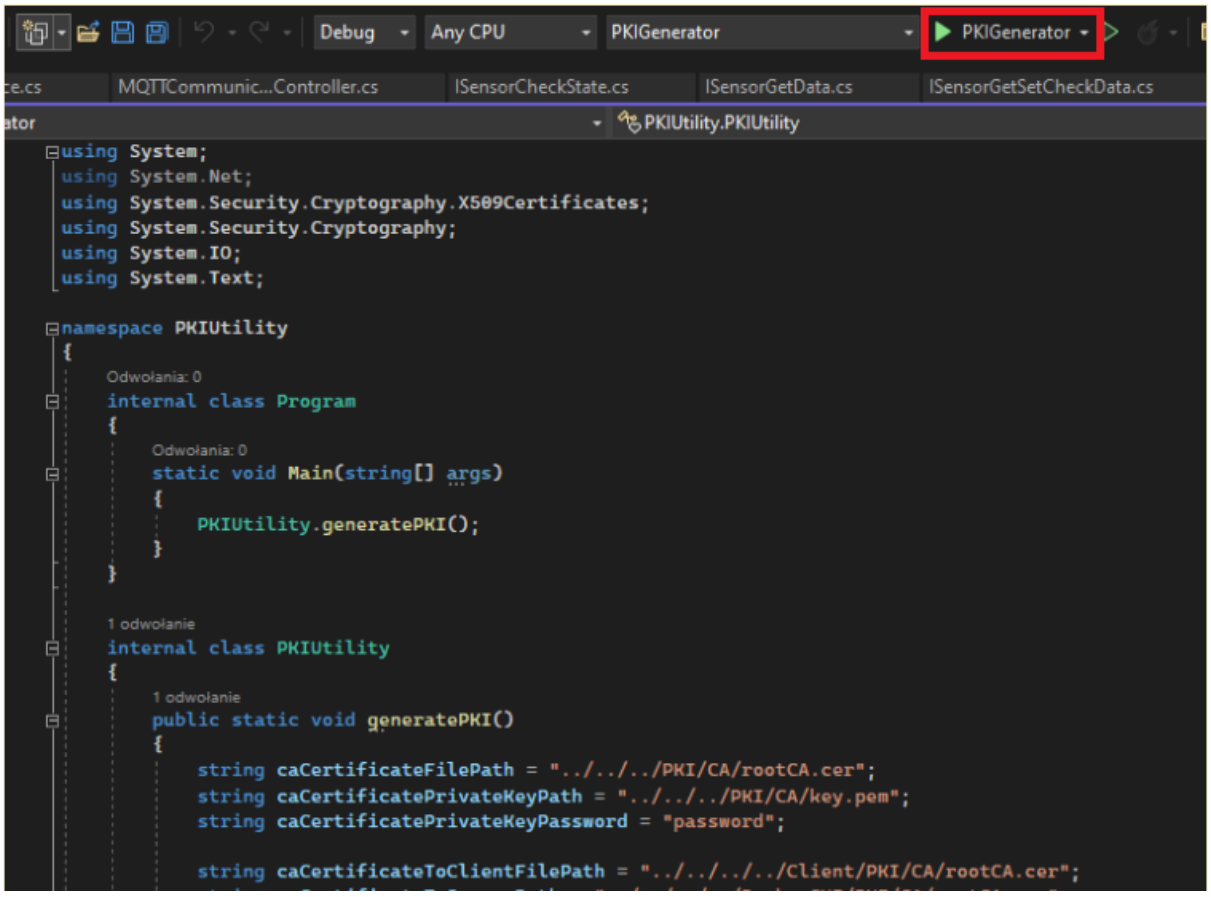
```
internal class Buzzer
    {
        // static GpioController controller;
        Process activeProcess;
        const string buzz_script = "Client/Sensors/square_wave.py";
        const string python_path = "/usr/bin/python3.9";
        bool state_;
        public Buzzer() {
            bool state_ = false;
            // States are supposed to be TRUE for buzzing and FALSE for no
buzzing (WITH GREAT LETTERS TRUE and FALSE (and are strings))
        }

        // Sets the buzzer on or off depending on parameter (true is on)
        // If trying to set to the same state as before, early return
        public void set(bool state)
        {
            Console.WriteLine("buzzer.set called");
            if(state == state_) return;
            state_ = state;
            if(state){
                send_square_wave();
            }
            if(!state){
                activeProcess.Kill();
            }
        }

        private void send_square_wave(){
            Console.WriteLine("Start python send sqr wave script");
            try{
                activeProcess = Process.Start(python_path, buzz_script);
            }
            catch(Exception e){
                Console.WriteLine(e.Message);
            }
        }
    }
```

Launching the alarm system

In order to launch the software you need to use dotnet and Visual Studio, in our case it's version 2022.

Steps:

1. Launch PKIGenerator software in Visual Studio in order to generate PKI infrastructure for your software



2. Compile broker software with Visual Studio and start it.

3. Start simulated client containers with Docker by using *run_simulated_client_docker_containers.ps1* Powershell script.

Setting up the Raspberry Pi client

In order to set up the Raspberry Pi to be used as a client, the following steps must be taken:

1. A Raspbian 32bit Lite OS (Bullseye) must be flashed on an SD card with at least 16 GB memory. This is best done using RaspberryPi imager, while setting up the wifi in the settings.
2. Install the necessary dependencies: [Dotnet for ARM single board computers] and setup the pythonnet dll and dotnet env variables
3. Export the client-specific environment variables

We've made steps 2 and 3 easy to do via the following bash script.

```
#!/bin/bash

# Clone the repo
sudo apt install git
git clone https://github.com/heyimjustalex/MQTTSmokeAlarmSystem.git

# Download and install dotnet
curl -sSL https://dot.net/v1/dotnet-install.sh > dotnet-install.sh
sudo chmod +x dotnet-install.sh
```

```
./dotnet-install.sh --channel 6.0

# Add necessary environment variables

# Find python3.9 library on the system
PYTHONLIBDIR=$(find / -name libpython3.9.so 2>/dev/null)
echo "found python 3.9 lib: ${PYTHONLIBDIR}"
echo "export PYTHONNET_PYDLL=${PYTHONLIBDIR}" >> ~/.bashrc

# Dotnet env variables
echo "export DOTNET_ROOT=$HOME/.dotnet" >> ~/.bashrc
echo "export PATH=$PATH:$HOME/.dotnet" >> ~/.bashrc

# Client-specific variables
echo "export USERNAME=client1" >> ~/.bashrc
echo "export CLIENT_ID=alarm1" >> ~/.bashrc
echo "export PASSWORD=password1" >> ~/.bashrc

# This needs to be individually adjusted to each sensor
echo "export BUZZER_THRESHOLD=0.02" >> ~/.bashrc

source ~/.bashrc
```

Setting up the pi can therefore be done by simply running `./setup_client.sh` in the console. This will download the repo into the directory of the script, so make sure you don't download the repo and run the script from there.

Building and running the client can then be done by launching `./run_client.sh`

Results

Outcome

We managed to deploy a working alarm system prototype that consists of:

- **broker software** - we managed to create a broker that is run by a Windows computer. The broker is able to pass messages to clients. Additionally the user interface does provide information about system state.

- **client software and hardware** - we managed to build a working hardware device that is able to use sensors to detect smoke and set off buzzers. The device runs client software and that's why it's able to send messages to the broker.

- **PKI generator -** utility project to generate and distribute certificates and keys

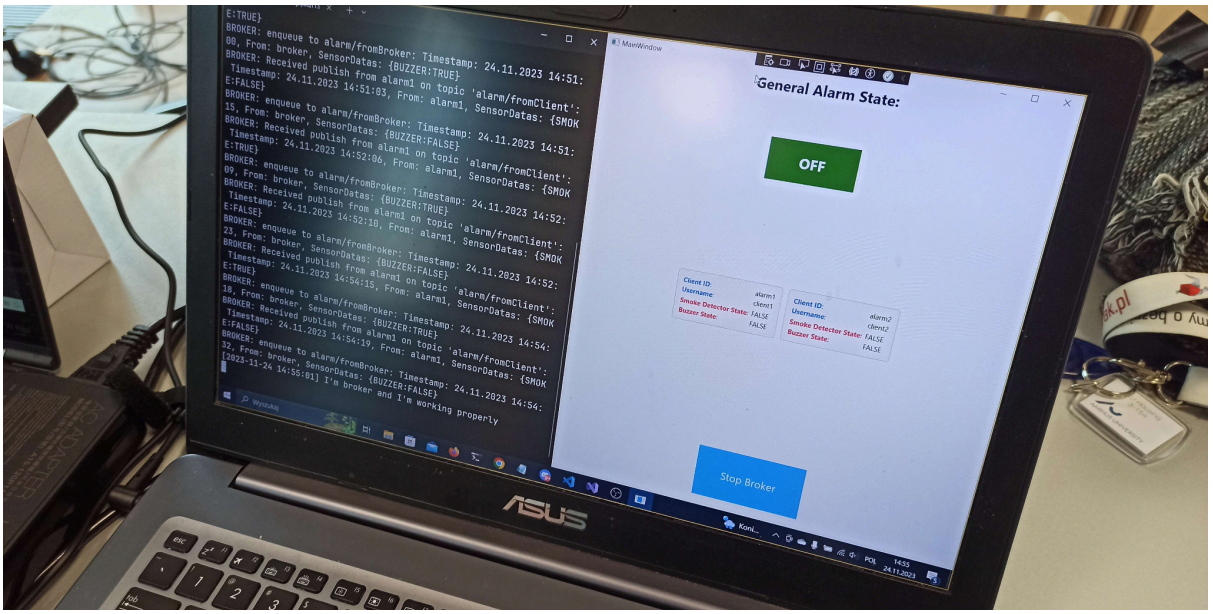


Figure 9 - Broker software with logs and GUI

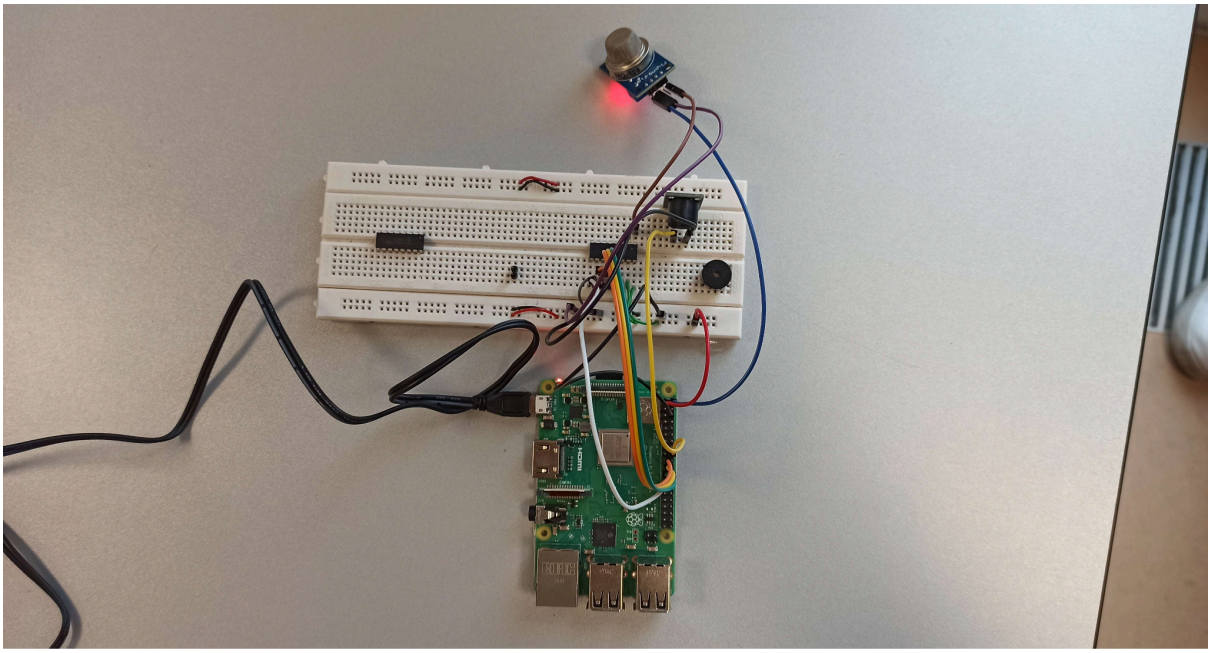


Figure 10 - Client hardware with buzzer and smoke detector

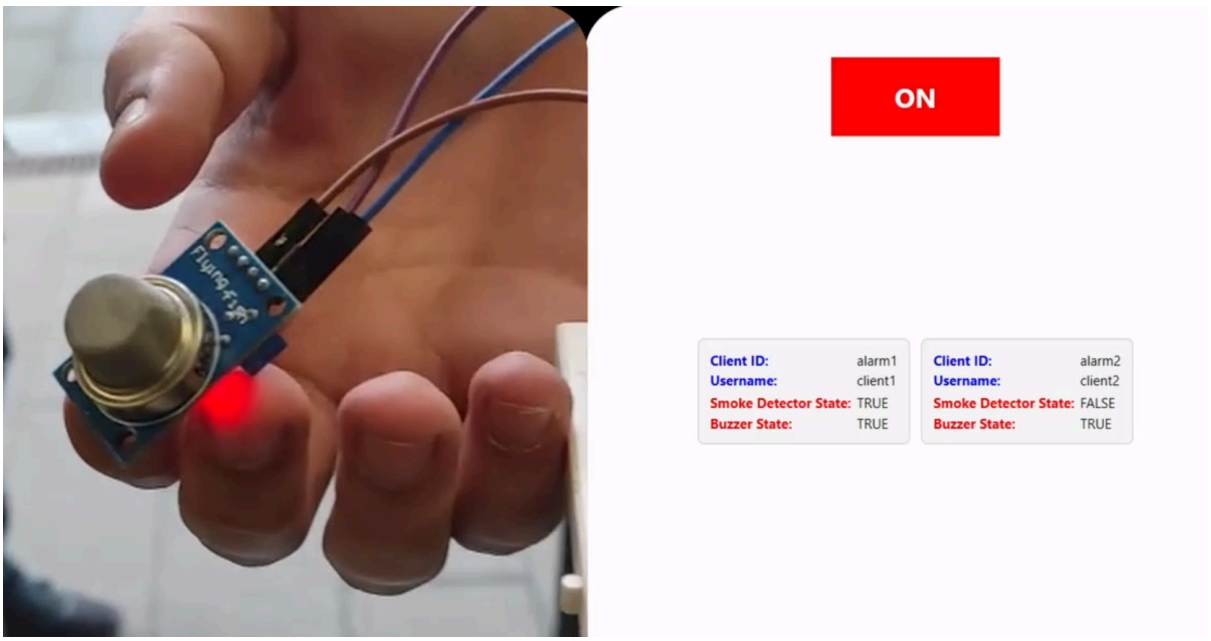


Figure 11 - Alarm that has been set off due to smoke detection

What has been successful

- **The alarm system is functional** - we are able to detect smoke, send message about the smoke to other clients and monitor the state of the system

- **Working implementation of mutual authentication** - for clients with usernames, passwords and for the broker with certificates. The MQTT over TLS works

- **We used real hardware** - we were able to cooperate on the intersection of two fields - hardware and software

- **The broker passes messages to the clients using MQTT protocol** - we were able to develop a solution which uses MQTTnet both for client and broker using pub/sub model and MQTT protocol

What has failed

- **Using WPF for UI made us unable to deploy the broker software to Linux** - it would be technically possible, however in general it would be better to write the UI part in different technology

- **The UI part is tightly coupled with broker software which should not be the case** - The UI/frontend should be decoupled as much as possible. We break separation of concerns and single responsibility principles

- **The deployment of PKI is complicated and not automatic** - If we had to revoke a certificate and deploy PKI with new certificates it would be a cumbersome process. Outside prototyping we should use some automation software to deploy PKI like Ansible or Jenkins

- **Using C# for client software was a bad idea** - we ended up using Python inside C#, with special C# library. It would be better for the client software to be implemented in Python, which works better with Raspberry PI.

Discussions and Challenges

Challenges:

- **Designing a proper architecture** - we made the design too early without the full understating of the topic

- **Designing and deploying PKI** - making a proper public key instructions with certificate chaining and combining it with MQTT broker was really challenging, but made us learn a lot

- **Getting RPI to work with analog sensor** - it was cumbersome to use C# for controlling IO and SPI interface, and it is not clear where the problems for SPI communications were. We solved that with python scripts and embedded python.

- **Setting up the RaspberryPI environment** - Since all the testing was done on a local network via mobile hotspot, every time the tests were run, the IP of the broker would change. We could solve that by using a static IP for the broker, or by using network scan tools like nmap to find it.

Conclusion

There are several conclusions and things that could be improved in our project:

- The architecture should be improved. The additional code of logic and UI is too coupled with the broker. We should adhere more to separation of concerns and single responsibility principle.

- We should consider adding external buzzers that are louder, which means that there should also be an external power source.

- The system could be more power efficient - using a RaspberryPi3 includes an entire OS and a lot of idle CPU usage. A more lightweight MCU or SoC might be preferable, such as ESP32, RPi nano/pico or others. Our prototype required only to control a buzzer, read from a sensor and send the data out via wireless technology.

- We did not have clear timing constraints. For a security system, we should at least spend some time figuring out how fast the system should react, and what could be the potential bottlenecks, as well as doing some tests.

- We should think of changing client technology to steer away from .NET, which is not supported by many MCU architectures, and has proven to be cumbersome when dealing with IO.

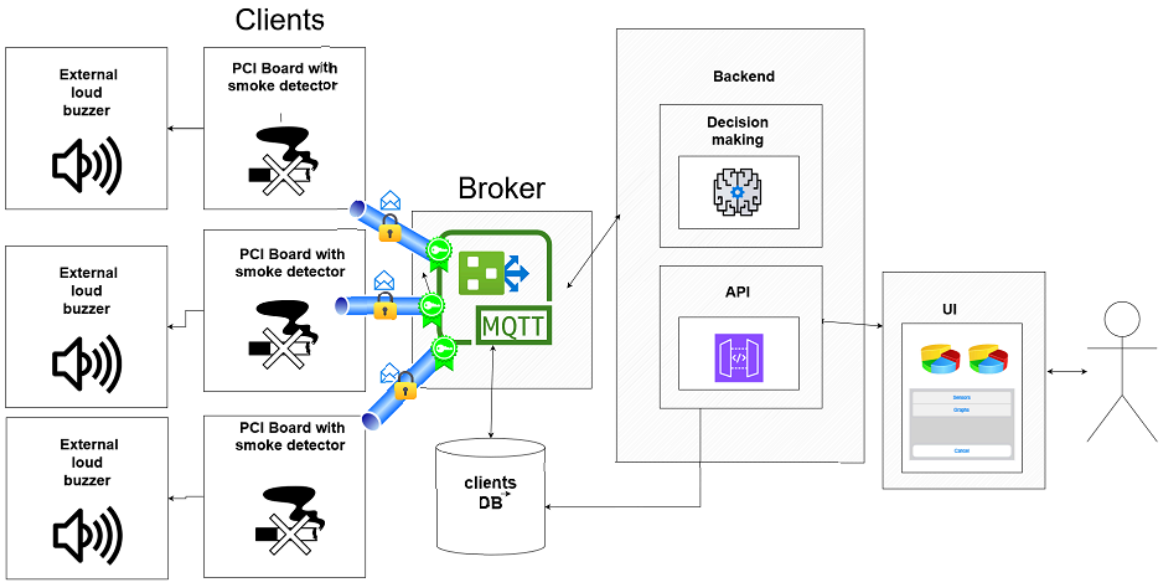


Figure 12 - Improved architecture of our alarm system

Future work would mean that we should probably start from scratch and reuse some components that we've built during the project (some code, PKI, hardware). Despite the mistakes we made, we've been able to deploy a functional solution and conclude what the architecture should look like when built in a real-world scenario.

The most important conclusion is however to put more work into the design of our solution and making sure that the assumptions and architecture are correct. The later you realize there are some bad assumptions the higher will be the cost of modifying them.

References:

https://github.com/dotnet/MQTTnet/issues
https://github.com/heyimjustalex/MQTTSmokeAlarmSystem
https://github.com/dotnet/MQTTnet/wiki/Client
https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.x509certificates.x509certificate?view=net-8.0
https://learn.microsoft.com/en-us/dotnet/iot/intro
https://learn.microsoft.com/en-us/dotnet/api/iot.device.adc.mcp3008?view=iot-dotnet-latest
https://github.com/pythonnet/pythonnet
https://stackoverflow.com/questions/6254703/thread-sleep-for-less-than-1-millisecond
*https://datasheets.raspberrypi.com/pico/pico-datasheet.pdf?_gl=1*8vsfnl*_ga*OTI5Mzk2MDQ3LjE2OTgzOTU5Njk.*_ga_22FD70LWDS*MTcwMjEyNzM2NS45LjAuMTcwMjEyNzM2NS4wLjAuMA..*
https://www.pidramble.com/wiki/benchmarks/power-consumption
https://csharp.hotexamples.com/examples/System.Security.Cryptography.X509Certificates/X509CertificateCollection/-/php-x509certificatecollection-class-examples.html
https://github.com/heyimjustalex/MQTTSmokeAlarmSystem/tree/main/Datasheets