

University of Alberta
Master of Science in Internetworking
Capstone Project (MINT 709)

IOT Enabled Natural Gas Detection Secure Wireless Sensor for residential and commercial deployments with cloud-based data storage

Supervisor

Gurpreet (Pete) Nanda

M.Sc. P.Eng. Sr.Solution Architect, Fujitsu Network
Communications, Richardson TX.

Author

Eduardo Oliva

Edmonton, February 2017

CONTENTS

ABSTRACT	3
KEY CONCEPTS AND INVOLVED TENCHNOLOGIES	4
THE DESIGN	8
A-ARDUINO WIRELESS GAS DETECTOR.....	9
B-LOCAL BROKER.....	11
C-ON LOCATION ACCESS POINT.	11
D-VMWARE ESXI SERVER AND VMWARE VSWITCH.....	11
E-VMWARE VM RUNNING CENTOS.	12
F-OPENSTACK NETWORKS, ROUTER, INSTANCE.....	12
THE SET-UP	14
A-BUILD AND SETUP THE WGS.....	14
1)GETTING READY:.....	14
2)THE IOT CODE:.....	15
3)THE SENSOR:.....	18
B-LOCAL BROKER.....	19
C-ACCESS POINT.....	21
D-VMWARE ESXI SERVER AND VMWARE VSWITCH	22
E-OPENSTACK VMWARE VM RUNNING WITH CENTOS SERVER 7.	22
F-USER PC AND/OR SMARTPHONE:	28
G-OPENSTACK NETWORKS, ROUTER, IMAGES AND INSTANCE.	28
H-REMOTE BROKER.....	32
I-SETUP SUMMARY AND PASSWORDS	35
THE SECURITY	37
INTERMEDIATE BRIDGE.....	38
POWER-ON	42

A-SERVER/INTERMEDIATE MOSQUITTO BROKER.....	42
B-CENTOS INSTANCE/MOSQUITTO REMOTE BROKER	44
C-THE LOCAL BROKER	44
D-THE WGD.....	45
E-GASDETECTOR APP AND MYSQL	46
F-TALK TO THE WGD.....	47
PERFORMANCE.....	49
FINAL CONSIDERATIONS	52
TO BE CONTINUED.....	54
APPENDIX A, BUDGET	55
APPENDIX B, TIMELINE	56

ABSTRACT

The detection of gas leaks is a major safety issue where citizens, employees and assets could suffer serious harm as the result of undetected malfunction or damage in facilities. Currently, most detection systems are expensive, passive, hard to wire, maintain and update. With every attempt to turn them into wireless or portable systems these problems are increased several times and can become impossible to achieve as well.

Whenever a potentially dangerous gas is involved (e.g. Natural Gas, CO₂, H₂S, etc), the responsible entities must satisfy safety regulations and protect human life. Utilities, Oil & Gas, Insurers and Building companies are the obvious and always interested parts in this equation where prevention is better (cheaper) than damage control.

IoT¹ is offering a wide span of reliable solutions with costs in a trend to the low. Automation and two ways communications are possible, then the times of isolated and passive detection systems are gone. Also, wireless communications have a positive impact in budgets and in the required time to install and setup devices.

IoT is a non-subversive revolution in development, that perfectly matches with virtualization and clouding environments and is spreading worldwide boosted by open source research and development. This research will reflect the foundations and spirit of the knowledge received during the MINT course. In order to achieve this high level of performance and professionalism, the information should be available always, on time and wherever is required as well. Sensors, wired and wireless networks, operating systems, internet security and programming languages understanding will be required to address the build the proposed system.

This project is a “Proof of Concept”, that will combine all the required technologies to achieve the desired objective (build the IoT system with Cloud computing and database services). Despite that currently there are already many commercial applications and more research in progress, some challenges might come from the specific combination selected for this project that will be used to start-up new concepts or verify the existing.

¹ Internet of The Things

KEY CONCEPTS AND INVOLVED TECHNOLOGIES

- **Arduino:**” is a computer hardware and software company, project, and user community that designs and manufactures microcontroller kits for building digital devices and interactive objects that can sense and control objects in the physical world. The project's products are distributed as open-source hardware and software, which are licensed under the GNU Lesser General Public License (LGPL) or the GNU General Public License (GPL).”²

Arduino is Based in the Wiring project, that was initiated in 2003 by the Colombian Hernando Barragan for his Master’s thesis at the Interaction Design Institute of Ivrea, Italy.

- **Arduino IDE:** The Arduino Integrated Development Environment - or Arduino Software (IDE) - contains a text editor for writing code, a message area, a text console, a toolbar with buttons for common functions and a series of menus. It connects to the Arduino and Genuino hardware to upload programs and communicate with them.³ The *Arduino* “language” is based on C/C++.
- **Broker (MQTT Broker):** is a Mosquitto Server, that depending on its proximity to the publishers and the use that gives to the received messages, could be Local or Remote. The Local Broker subscribes messages from the publishers in a LAN, and forwards these messages to a Remote Broker. A Remote Broker may receive messages though the internet, from Local Brokers or directly from the publishers. The Remote Broker is the destination of the messages.
- **Bridge (MQTT Bridge):** a Mosquitto Bridge can be created between two brokers, a local and a remote broker. While some devices might be publishing messages to a local broker, it could be bridged to a remote broker that will receive the messages. If the local broker has the right user and password, the messages that were supposed to be received by the local broker, will be published in the remote broker. The brokers concept should be understood as “servers”.
- **CentOS:** “The CentOS Project is a community-driven free software effort focused on delivering a robust open source ecosystem. For users, we offer a consistent manageable platform that suits a wide variety of deployments. For open source communities, we offer a solid, predictable base to build upon, along with extensive resources to build, test, release, and maintain their code.”⁴
- **Dashboard:** the OpenStack Dashboard is the tool that a cloud end user can use to provision his resources in the limits that the administrator has set up for that user. Is a web based interface that includes all the OpenStack Services, in some OpenStack deployments is known as Horizon.
- **ESP8266:** “The ESP8266 Wi-Fi Module is a self-contained SOC with integrated TCP/IP protocol stack that can give any microcontroller access to your Wi-Fi network. The ESP8266 is capable of either hosting an application or offloading all Wi-Fi networking functions from another application processor. Each ESP8266 module comes pre-programmed with an AT command set firmware, meaning, you can simply hook this up to your Arduino device and get about as much Wi-Fi-ability as a Wi-Fi Shield offers (and that’s just out of the box)! The ESP8266 module is an extremely cost effective board with a huge, and ever growing, community.”⁵
- **Iaas:** Infrastructure as a service.

² <https://en.wikipedia.org/wiki/Arduino>

³ <https://www.arduino.cc/en/guide/environment>

⁴ <https://www.centos.org/>

⁵ <https://www.sparkfun.com/products/13678>

- **IoT**; The **Internet of things (Internet of Things or IoT)** “is the internetworking of physical devices, vehicles (also referred to as "connected devices" and "smart devices"), buildings, and other items—embedded with electronics, software, sensors, actuators, and network connectivity that enable these objects to collect and exchange data.”⁶
- **Instance**: an OpenStack instance is the equivalent to a Virtual Machine.
- **Mosquitto**: “is an open source (EPL/EDL licensed) message broker that implements the MQTT protocol versions 3.1 and 3.1.1. MQTT provides a lightweight method of carrying out messaging using a publish/subscribe model. This makes it suitable for "Internet of Things" messaging such as with low power sensors or mobile devices such as phones, embedded computers or microcontrollers like the Arduino.”⁷

Mosquitto shares information using a Publisher/Subscriber system. For publishing a message, the publisher does not need a subscriber waiting and it does not require to wait for acknowledgement to publish the next message. The subscriber must be waiting to receive messages and will listen only to the “topics” it is subscribing. Mosquitto uses the ports 1883 and 8883. The port 8883 uses SSL and the messages are encrypted. Mosquitto also supports TLS. An IoT device can publish messages in a broker, and also can subscribe messages from them or from other publishers.

- **MySQL**: “is an open-source relational database management system (RDBMS). Its name is a combination of "My", the name of co-founder Michael Widenius' daughter, and "SQL", the abbreviation for Structured Query Language. The MySQL development project has made its source code available under the terms of the GNU General Public License, as well as under a variety of proprietary agreements. MySQL was owned and sponsored by a single for-profit firm, the Swedish company MySQL AB, now owned by Oracle Corporation. For proprietary use, several paid editions are available, and offer additional functionality.”⁸
- **MQ Gas Sensor**: “The MQ series of gas sensors use a small heater inside with an electro-chemical sensor. They are sensitive for a range of gasses and are used indoors at room temperature. They can be calibrated more or less, but a concentration of the measured gas or gasses is needed for that.”⁹ MQ Gas Sensors use the same principle that Naoyoshi Taguchi employed in 1962 to build the first semiconductor device that could detect gases. There are different versions for different applications, and the most common are numbered MQ-2 to MQ-9 being able to detect: Combustible gases and Smoke, Alcohol, Natural Gas-Methane, LPG-Natural Gas-Coal Gas, LPG-Propane, Carbon Monoxide, Hydrogen, CO2 and combustible gas.
- **Open-source model**: “is a decentralized development model that encourages open collaboration. A main principle of open-source software development is peer production, with products such as source code, blueprints, and documentation freely available to the public. The open-source movement in software began as a response to the limitations of proprietary code. The model is used for projects such as in open-source appropriate technologies, and open-source drug discovery.”¹⁰

⁶ https://en.wikipedia.org/wiki/Internet_of_things.

⁷ <https://mosquitto.org/>

⁸ <https://en.wikipedia.org/wiki/MySQL>

⁹ <http://playground.arduino.cc/Main/MQGasSensors>

¹⁰ https://en.wikipedia.org/wiki/Open-source_model

- **OpenStack:** “OpenStack is a free and open-source software platform for cloud computing, mostly deployed as an infrastructure-as-a-service (IaaS).^[3] The software platform consists of interrelated components that control diverse, multi-vendor hardware pools of processing, storage, and networking resources throughout a data center. Users either manage it through a web-based dashboard, through command-line tools, or through a RESTful API. OpenStack.org released it under the terms of the Apache License.”¹¹
- **OpenStack Components:** a basic set of component should include:

Service(Name)	Function
Compute (Nova)	Is the computing controller and is the main part of an IaaS.
Networking (Neutron)	Manages networks and IP address.
Block Storage (Cinder)	Block level storage for OpenStack instances.
Identity (Keystone)	Central directory of users mapped to OpenStack services they can access
Image (Glance)	Images are templates to deploy volumes (disks) and instances (VMs)
Object Storage (Swift)	Scalable redundant storage.
Dashboard(Horizon)	Graphical interface for administrators and users.
Orchestration (Heat)	Orchestrates multiple composite cloud applications using templates
Telemetry (Ceilometer)	Single point of contact for billing system.

Table-1. OpenStack Services

Other OpenStack services are: WorkFlow (Mistral), Database (Trove), Bare Metal (Ironic), Messaging (Zaqar), Shared File System(Manila), DNS (Designate), Search (SearchLight), Key Manager (Barbican).

- **Packstack:** “is a utility that uses Puppet modules to deploy various parts of OpenStack on multiple pre-installed servers over SSH automatically. Currently only CentOS , Red Hat Enterprise Linux (RHEL) and compatible derivatives of both are supported.”¹²
- **Puppet:** “Puppet Enterprise is the leading platform for automatically delivering, operating and securing your infrastructure – no matter where it runs. With Puppet you know exactly what is going on with all your software. And you get the automation needed to drive change with confidence.”¹³
 “Puppet provides a standard way of delivering and operating software, no matter where it runs. With the Puppet approach, you define what you want your apps and infrastructure to look like using a common easy-to-read language. From there you can share, test and enforce the changes

¹¹ <https://en.wikipedia.org/wiki/OpenStack>

¹² <https://wiki.openstack.org/wiki/Packstack>

¹³ <https://puppet.com/product>

you want to make across your datacenter. And at every step of the way, you have the visibility and reporting you need to make decisions and prove compliance.”¹⁴

- **TLS**: “Transport Layer Security (**TLS**) and its predecessor, Secure Sockets Layer (SSL), both frequently referred to as “SSL”, are cryptographic protocols that provide communications security over a computer network.”¹⁵
- **Wi-Fi**: “Wi-Fi or Wi-Fi is a technology for wireless local area networking with devices based on the IEEE 802.11 standards. Wi-Fi is a trademark of the Wi-Fi Alliance, which restricts the use of the term Wi-Fi Certified to products that successfully complete interoperability certification testing.”¹⁶
- **VMware ESXi**: “VMware Vsphere hypervisor is a free bare-metal hypervisor that virtualizes servers so you can consolidate your applications on less hardware.”¹⁷
“VMware ESXi is the industry-leading, purpose-built bare-metal hypervisor. ESXi installs directly onto your physical server enabling it to be partitioned into multiple logical servers referred to as virtual machines. Customers can use ESXi with either the free vSphere Hypervisor or as part of a paid vSphere edition.”¹⁸

¹⁴ <https://puppet.com/product/how-puppet-works>

¹⁵ https://www.google.ca/search?q=tls&ie=utf-8&oe=utf-8&gws_rd=cr&ei=BnesWM-jF5DejwOv_4DYDw

¹⁶ <https://en.wikipedia.org/wiki/Wi-Fi>

¹⁷ <http://www.vmware.com/ca/products/vsphere-hypervisor.html>

¹⁸ <http://www.vmware.com/products/esxi-and-esx.html>

THE DESIGN

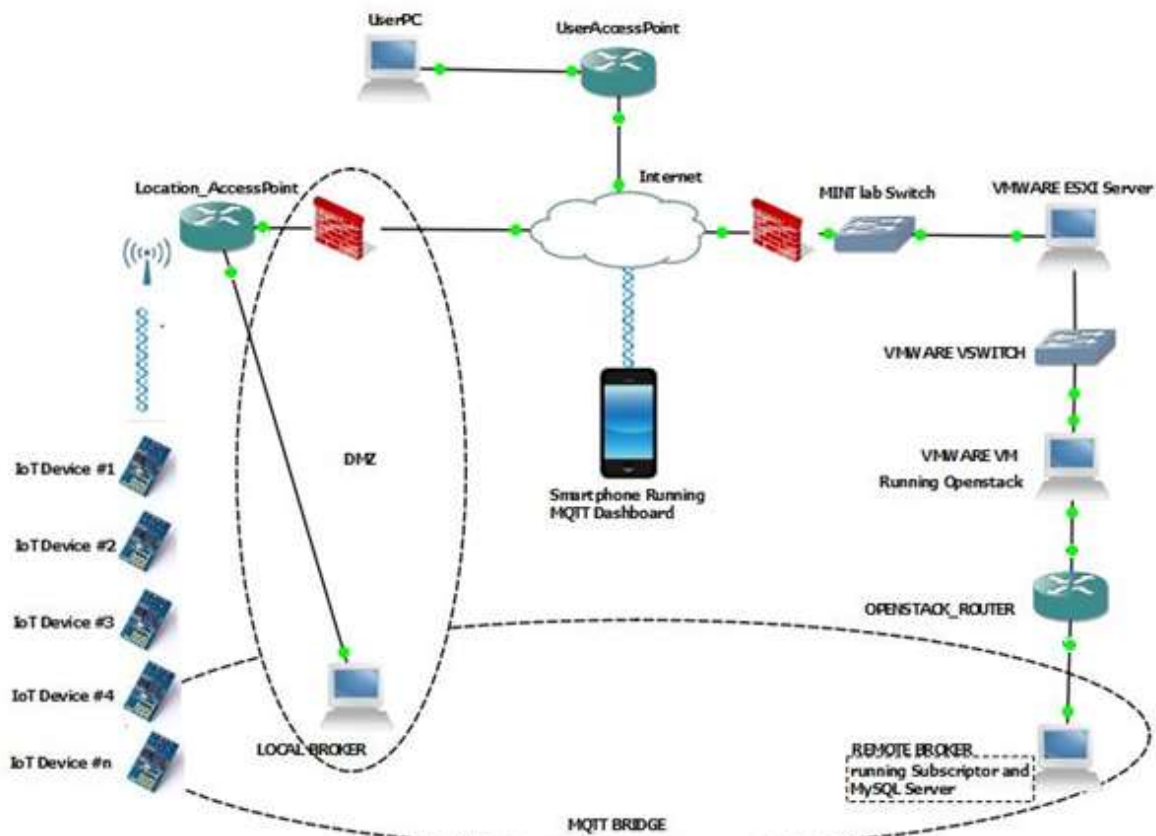


Figure-0 The System's Design

The main objective is to build a system that should produce information from IoT devices and store it into a Cloud Based system. The involved devices (actual and virtualized devices) that we will be setting up are:

- A. IoT Arduino Wireless Gas Detector (WGD).
- B. Local Broker.
- C. On Location Access Point.
- D. VMWare ESXi Server and VMWare Vswitch.
- E. CentOS VMWare VM running Centos.
- F. OpenStack Networks, Router, Images and Instance. (Remote Broker)

An optional component of this system could be the integration of any Smartphone and UserPC connected to Internet that could be used to interact with the IoT devices. Their presence depends of security considerations that will be discussed in the following sections of this report.

Following, a brief description of each one is provided, but more detailed explanation about features and set-up will be developed in a separated section.

A-ARDUINO WIRELESS GAS DETECTOR

The first device, and source of the information, of this system is the “Wireless Gas Detector” (WGD) built with Arduino compatible parts. The basic design includes the Sensor, Wi-Fi adapter, a couple of Resistors, a Capacitor and a Transistor.

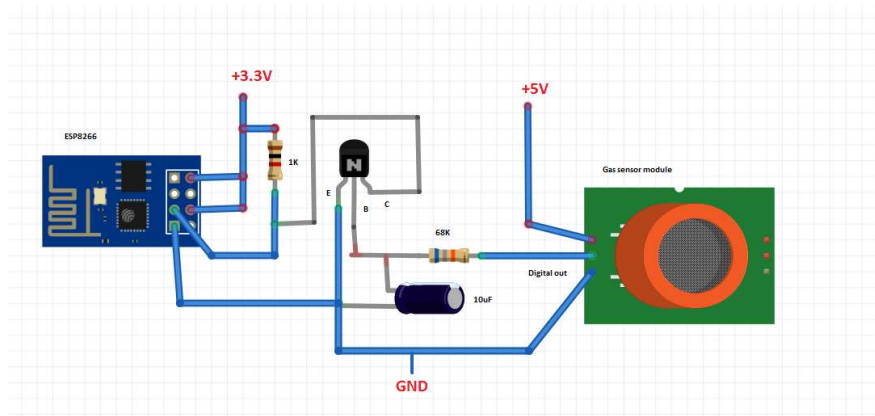


Figure-1 Circuit Design¹⁹

The actual circuit is shown in Figure-2, and includes another capacitor and accepts a Serial Adapter. Three capacitors were added because ESP8266-01 requires power during the connection to the AP. Without them it reboots with error message #3, at rate of two out of every three times you power on the circuit. The solution is to add a couple of capacitors to the 3.3V power source and one more (smaller) in the closest connection to the power intake of ESP8266-01. The solution provides a flawless booting and connecting performance.

¹⁹ <http://iot-playground.com/blog/2-uncategorised/53-esp8266-wifi-gas-sensor-arduino-ide>

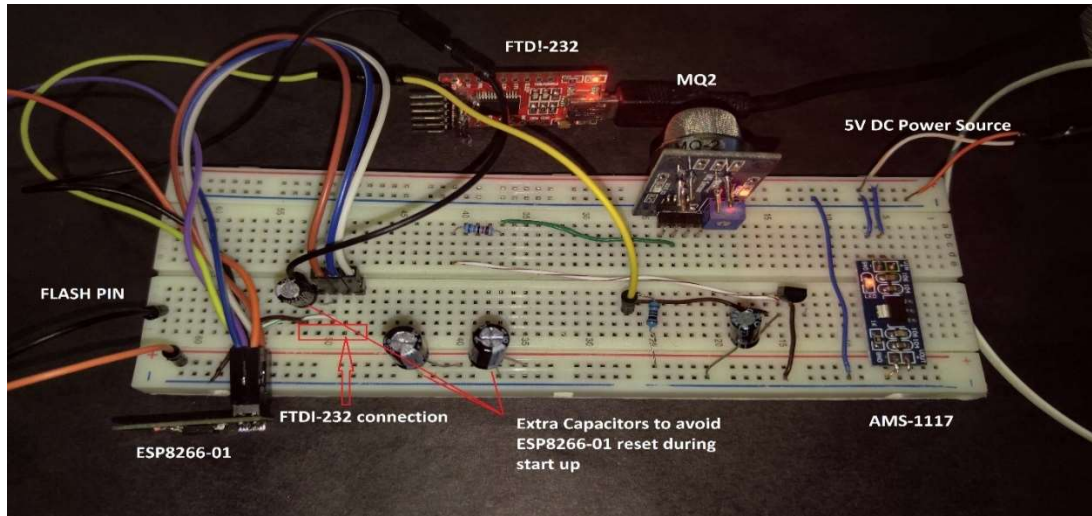


Figure-2. Wireless Gas Detector (WGD)

Part	Features
ESP8266-01	Provides Wi-Fi (802.11 b/g/n) connection. Serial/UART baud rate: 115200 bps. Integrated TCP-IP protocol Stack. Input power: 3.3 V. Memory 1MB. Dimensions 24.75mm x 14.5mm.
MQ2	Circuit Voltage 5V DC. Digital and Analog output. Sensitive to LPG, butane, propane, methane, alcohol, hydrogen and smoke. Heating time 3-5min.
AMS 1117	Low dropout voltage regulator. Input power: 5V DC. Output current 1A. 3.3V DC output.
FTDI 232	USB to RS-232 converter (used to load the app to Esp8266-01 and connect the serial-monitor)

Table-2. Wireless Gas Detector Components

To power on the device, this could be done in two different ways:

- 1- Connect a regulated 5V DC power source as indicated in Figure 2. This will power on the whole system, including the Gas Sensor and the Wi-Fi ESP8266 device.
- 2- Connect the FTDI232 Serial to USB adapter, to the “FTDI-232 connection” pins. This will power the ESP8266 only. The gas sensor won’t be energized and therefore the Input Reading that ESP8266 will receive will be always “1” (gas presence).

A third and proven alternative is using both, 5VDC and FTDI232 at the same time. This is only useful if you want to energize the sensor while you are reading the outputs of the ESP8266 serial port in Arduino’s IDE Serial Monitor. In this case the default reading will be “0” (no gas). In this case you must disconnect the AMS1117 Regulator to avoid malfunction or damaging the parts.

The manufacturers are explicit about not energizing the pieces with other voltage than the recommended. The developed experience is that the parts are more reliable than expected.

B-LOCAL BROKER

The Local Broker is the On-Location Server. It is built on a Gateway 2000 laptop model W35OI (Celeron processor with 2GB RAM and 160GB Hard Drive) running Ubuntu 14.04. This could be replaced with a low cost Raspberry Pi module or similar equipment as it is not intended to save any data or perform any advanced function either. Its main function is to establish the MQTT Bridge with the Remote Server. During the development phase, it was used to test publishing and subscribing messages with the WGS.

C-ON LOCATION ACCESS POINT.

A connection with Internet is required to be able to connect to the Remote Broker. For this project, the ISP was Telus and the AP model was V1000H.

D-VMWARE ESXI SERVER AND VMWARE VSWITCH.

VMware vSphere 6 Enterprise is installed in a PowerEdge R430 server installed in the University of Alberta Mint Lab. A tunnel connection is used to access it with VMware vSphere Client.

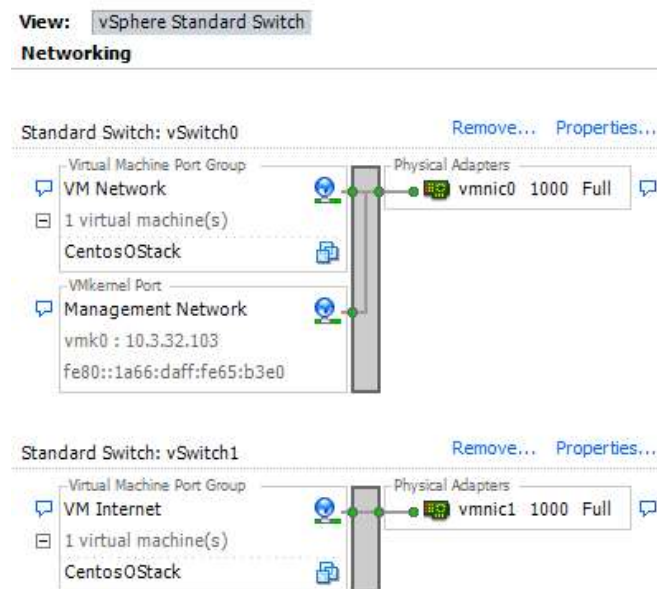


Figure-3 VMware vSwitch Configuration

To connect to the server, VMware vSwitches are used: vSwitch0 to provide management connection (through the tunnel) and vSwitch1 to have connection to internet through a Public IP address provided by the UofA Mint Lab.

E-VMWARE VM RUNNING CENTOS.

A Virtual Machine running CentOS 7 Server was deployed in VMware (called CentOStack). This machine has two NICs (one for management and another one for internet), 30 CPUs, 38GB RAM and 200GB Hard Drive among most important features.

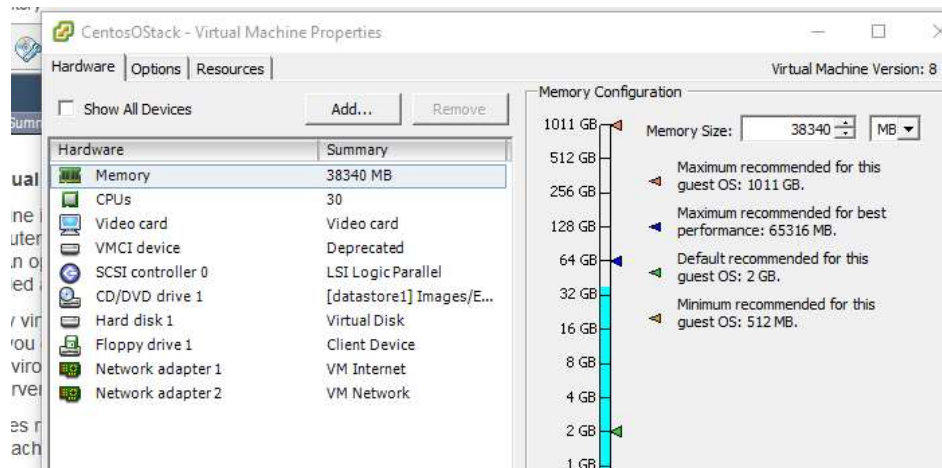


Figure-4 CentOStack VM settings

F-OPENSTACK NETWORKS, ROUTER, INSTANCE.

In the CentOStack VM, PackStack OpenStack was deployed. In consequence, some virtualized devices were created. The most relevant are:

- Networks: two networks were created, Public and Internal. Public is to get connection to the “outside” world, though the CentOStack “br-ex” interface. Internal is the network that connects to all the OpenStack Instances that could have the project.
- Router: A router is created in OpenStack to connect the Public Network and the Internal Network.

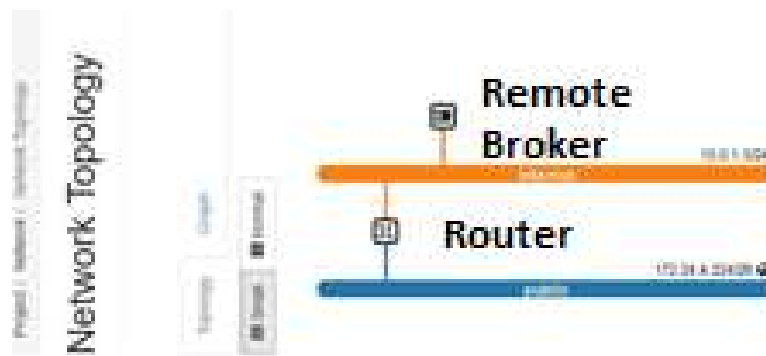


Figure-5 OpenStack Network Topology.

- Instance: is the equivalent to a Virtual Machine, its name is GD01. The OS is CentOS server v7 installed using one modified OpenStack Image and a CentOS ISO OpenStack image. This instance has the following features:
 - 4 Processors.
 - 8GB RAM.
 - 30GB Hard Drive.
 - 1 NIC.



The screenshot shows the OpenStack dashboard interface. On the left, there is a sidebar with navigation links: Volumes, Images, Access & Security, NETWORK, and OBJECT STORE. The main content area displays a table of instances. The table has columns for Instance Name, Image Name, IP Address, Size, Key Pair, Status, Availability Zone, Task, and Power State. A single instance named 'GD01' is listed with an IP address of 10.0.1.8, size of m1.large, key pair of GasKey, status of Shutoff, availability zone of nova, task of None, and power state of Shut Down. Below the table, it says 'Displaying 1 item'.

Instance Name	Image Name	IP Address	Size	Key Pair	Status	Availability Zone	Task	Power State
GD01	-	10.0.1.8	m1.large	GasKey	Shutoff	nova	None	Shut Down

Figure-6. GD01 Instance in OpenStack.

The OpenStack Instances will be responsible of providing Iaas (computing) and Daas (data storage) for the Remote Broker. This Broker is the Mosquitto entity that processes and stores all the data generated by the WGD.

THE SET-UP

To describe the entire system installation and setup process, we will go through the same list of components we identified in the previous section.

A-BUILD AND SETUP THE WGS

1)GETTING READY:

As ESP8266 libraries are not included in Arduino IDE default setup, you must add it to the active libraries. After this step, you will be able to use the ESP8266Wi-Fi.h. This is also a good time to add the MQTT library to your IDE (PubSubClient.h) and Base64.h that will be useful to convert some data types in the next steps.

Following Figure 1, the circuit was built on a “bread board”. The first time you connect the power source, the ESP8266 might disable your Wi-Fi network. This is because it has no pre-installed firmware. To solve this issue you can use a simple example code for Wi-Fi (like “Basic ESP8266 MQTT example”), to upload the ESP8266-01 configuration. Basically, all you have to do is include you SSID name, User and Password. Once you upload this basic configuration it will stop avoiding your Wi-Fi to work properly (It doesn’t matter if you provide inaccurate information or the AP is not powered on at this time).

For this task you must connect the Serial FTDI232 adapter, and plug the Flash Pin cable to GND connector (see yellow line showing where should this pin be connected in Figure-2). In the Arduino IDE Serial Monitor you should be able to see the ESP8266 trying to connect to an AP using the parameters you provided (If using our code there are some outputs to Serial Monitor to show this process evolution).

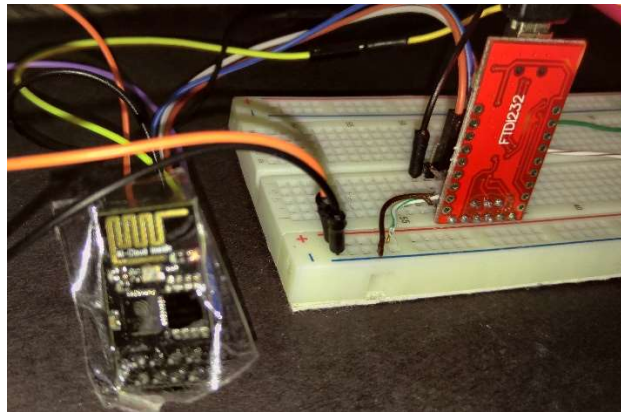


Figure-7. Connection to ESP8266 Serial Port.

Most of the times, the ESP8266 will reset by itself after loading a new code, and it will start working even when the Flash Pin is connected. For better results, it is recommended to remove the Flash Pin and reset the entire WGD(disconnecting the power source and connecting it again). Figure-8 can provide a better understanding of the FTDI232 wiring.

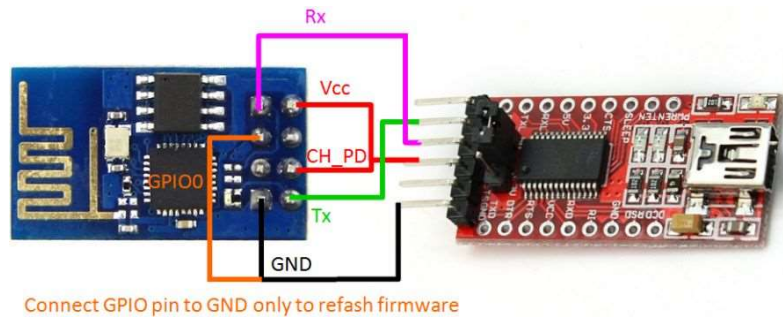


Figure-8. FTDI232 pin connection to ESP8266-01.²⁰

Once you uploaded your first code to ESP8266, you should be able to see attempts to connect to the AP in the serial monitor. It is important to don't try to open the Serial Monitor while uploading a program to your device, because this will stop the ongoing process. Figure-9 shows a successful uploading process.

```
Done uploading.

Sketch uses 233,309 bytes (53%) of program storage space. Maximum is 434,160 bytes.
Global variables use 32,772 bytes (40%) of dynamic memory, leaving 49,148 bytes for local variables. Maximum is 81,920 bytes.
Uploading 237456 bytes from /tmp/arduino_build_604920/capstoneGasDetector.ino.bin to flash at 0x00000000
..... [ 34% ]
..... [ 68% ]
..... [ 100% ]
```

Figure-9 Uploading code to ESP8266 through FTDI232



Figure-10 ESP8266, Arduino's IDE Serial Monitor shows Wi-Fi is successfully connected to AP

2)THE IOT CODE:

Code-1 shows the actual version that was uploaded to ESP8266.

²⁰ <http://www.instructables.com/id/ESP8266-based-web-configurable-wifi-general-purpos/>

CODE	Comments
<pre> #include <PubSubClient.h> #include <ESP8266Wi-Fi.h> #include <Base64.h> #define INPUT_PIN 2 // Update these with values suitable for your network. const char* ssid = "TELUS1977"; const char* password = "6"; byte mac[6]; Wi-FiClient espClient; PubSubClient client(espClient); long lastMsg = 0; char msg[140]; int value = 0; int ttime=1; String strtime; char myIpString[24]; char myTopic[40]; char myMACADDRESS[30]; const char* DetectorType="Gas"; void setup() { //pinMode(BUILTIN_LED, OUTPUT); // Initialize the BUILTIN_LED pin as an output Serial.begin(115200); pinMode(INPUT_PIN, INPUT); Serial.print("Starting execution, next is setup Wi-Fi"); setup_Wi-Fi(); Wi-Fi.macAddress(mac); client.setServer(IPAddress(192,168,1,16), 8883); client.setCallback(callback); } void setup_Wi-Fi() { delay(10); // We start by connecting to a Wi-Fi network Serial.println("BEGIN SETUP"); Serial.print("Connecting to "); Serial.println(ssid); //macString==mac2String(mac);// Serial.print("MAC address:"); byte myMAC[6]; Wi-Fi.macAddress(myMAC); sprintf(myMACADDRESS, "%02X:%02X:%02X:%02X:%02X:%02X", myMAC[5],myMAC[4],myMAC[3],myMAC[2],myMAC[1],myMAC[0]); Serial.println(myIpString); Wi-Fi.begin(ssid, password); while (Wi-Fi.status() != WL_CONNECTED) { delay(50); Serial.print("."); } } </pre>	<p>Your AP configuration.</p> <p>ESP8266 Initialization.</p> <p>Serial port speed. Data input pin. Serial.print sends data to Serial Monitor useful during configuration and test Get IoT device's MAC address. Set MQTT server IP. Callback initialized to be waiting for incoming msgs.</p> <p>Wi-Fi device Setup.</p> <p>Convert MAC address to String</p> <p>Try connection to AP If connection fails retry.</p>

<pre> Serial.println(""); Serial.println("Wi-Fi connected"); Serial.println("IP address: "); Serial.println(Wi-Fi.localIP()); IPAddress myIp = Wi-Fi.localIP(); sprintf(myIpString, "%d.%d.%d.%d", myIp[0], myIp[1], myIp[2], myIp[3]); //Spark.variable("ipAddress", myIpString, STRING);// } void callback(char* topic, byte* payload, unsigned int length) { char message_buff[100]; Serial.print("Message arrived ["); Serial.print(topic); Serial.print("] "); for (int i = 0; i < length; i++) { Serial.print((char)payload[i]); message_buff[i] = payload[i]; } String topic_str=String(topic); //Serial.println("Before storing msg"); String msgString = String(message_buff); //Serial.println("After storing msg"); if (msgString.startsWith ("Request")) { if (topic_str.endsWith(myIpString)) { //Serial.println("request comparisson succeed"); snprintf(myTopic,40,"outTopic/23.17.224.11/%s", myIpString); snprintf (msg, 140,"REQUESTED-%ld-%s Sensor IP:%s MAC:%s Reading value=%ld",value,DetectorType,myIpString,myMACADDRESS,digitalRead(INPUT_PI N)); Serial.print("Publish message: "); Serial.println(msg); client.publish(myTopic, msg); } } if (msgString.startsWith ("Sleep")) { if (topic_str.endsWith(myIpString)) { strtime=(msgString.substring(5,msgString.length()-1)); ttime=(strtime.toInt()); Serial.println(ttime); } } } void reconnect() { // Loop until we're reconnected while (!client.connected()) { Serial.print("Attempting MQTT connection..."); // Attempt to connect if (client.connect("ESP8266Client","user1","silvana01")) { Serial.println("connected"); </pre>	<p>Get IP address assigned by DHCP.</p> <p>Callback function analyzes publishers' attempts to publish in this device. Only receiving subscribed topics</p> <p>The payload is stored in a msg buffer.</p> <p>Only two types of msgs are of our interest: 1-Requests, for a sensor reading in this moment. The payload says "Request" 2-Modifications to the sleep function. The payload must contain "Sleepn-" where "n" Is the delay time between two sensor readings.</p> <p>23.17.224.11 is the IP assigned by the ISP.</p> <p>MQTT reconnect will loop while v connected to the AP.</p> <p>The Local Broker's user and password</p>
---	--

<pre> // Once connected, publish an announcement... sprintf(myTopic,"outTopic/23.17.224.11/%s", myIpString); snprintf(msg,140,"IOT %s sensor prompt, HELLO!", DetectorType); //Serial.println(myTopic); //Serial.println(msg); client.publish(myTopic, msg); // ... and resubscribe snprintf(myTopic,80,"InTopic/23.17.224.11/%s", myIpString); client.subscribe(myTopic); } else { Serial.print("failed, rc="); Serial.print(client.state()); Serial.println(" try again in 2 seconds"); // Wait 2 seconds before retrying delay(2000); } delay(1000); } } void loop() { if (!client.connected()) { reconnect(); } client.loop(); long now = millis(); if (now - lastMsg > (1000* ttime)) { lastMsg = now; ++value; snprintf(myTopic,40,"outTopic/23.17.224.11/%s", myIpString); Serial.println(myTopic); snprintf (msg, 140,"%ld-%s Sensor IP:%s MAC:%s Reading value=%ld", value,DetectorType,myIpString,myMACADDRESS,digitalRead(INPUT_PIN)); Serial.print("Publish message: "); Serial.println(msg); client.publish(myTopic, msg); delay(1000); } } } </pre>	<p>are provided to connect. (the ones stored and encrypted in etc/mosquitto/passwd file) The payload and topic are built.</p> <p>Main section of the code.</p> <p>Will loop forever.</p> <p>Print to Serial Monitor outgoing topics and messages (payloads). If a Sleepn- message is received, the delay time between sent messages is multiplied n times (ttime variable).</p>
---	---

Code-1. CapstoneGasDetector.ino

3)THE SENSOR:

As already described, the sensor is an analog device that generates a resistance in presence of certain gases.

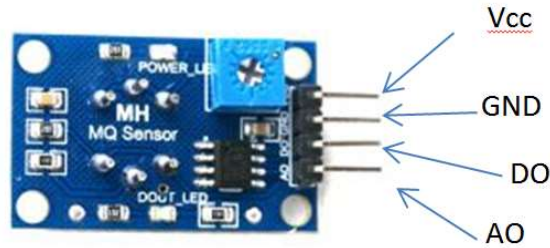


Figure-11. MQ2 connection pins.²¹

Vcc is connected to the 5VDC, while GND to ground. DO is the digital output (1 or 0) and AO is the analog output that can be adjusted using the included blue dimmer in the back side (see Figure 11). In this project we will use DO. The sensor doesn't accept more configuration than adjusting the AO range. Although, it requires a "warm up" period (2-3min) to start giving valid readings. Such delay should not result in any lack of accuracy, because the ESP8266-01 booting process, Wi-Fi connection and MQTT client connection process could take the same or more time to succeed (depending on environmental conditions of the radioelectric spectrum).

B-LOCAL BROKER

One of the most interesting features of IoT devices, is that they can connect directly to the Internet without any more intermediate devices than an Access Point. Although, they are also interesting because they can receive messages from the internet as well. But at the same time they are not supposed to work by themselves, and where one IoT device is installed, it is possible to have a large series of similar ones performing the same task, for redundancy or to cover a larger surface. Nevertheless the reason to have many IoT devices in the same place, it is not possible to provide one public IP address to each one. This is the first reason to use a Local Broker.

The other reason to have a Local Broker, is because IoT devices have limited memory and processing power, and in consequence they are very exposed to security breaches. That being said, a Mosquitto Broker may perform as intermediary between the IoT devices and the main server or Remote Broker, creating a secured bridge (the security will have a separate discussion in another section).

The local Broker is the known reference for all the IoT devices in the LAN. It was installed in a low cost portable device, with Ubuntu 14.04 as Operating System. The installed protocol was Mosquitto 3.1.

Mosquitto.conf file	Comments
allow_anonymous false password_file /etc/mosquitto/passwd port 8883 connection BridgeIt bridge insecure false	Don't allow anonymous publishers Location of the passwords file (local) user by Arduino code to connect. Port to use (SSL secured)

²¹ <http://www.instructables.com/id/ESP8266-based-web-configurable-wifi-general-purpos/>

cleansession false	Name of bridge connection to Remote Broker
clientid br_local	No insecure connection to Remote Broker
start_type automatic	
remote_username user1	Username to connect to Remote Broker
remote_password silvana01	Password to connect to Remo
address 129.128.116.190:8883	IP of the Remote Broker and port to connect.
topic outTopic/# both	Topics to be used in this bridge.
topic InTopic/# both	

Code-2 Local Broker, configuration file /etc/mosquitto/mosquitto.conf

In the Arduino Code (Code-1), the IP address that is provided for connection is 192.168.1.16, that belongs to the Local Broker. Therefore, ESP8266 connects with it and publishes its topics in the Local Broker. But, because of the Bridge, all the subscribed topics that correspond with the ones in the declared in the bridge, will be published in the Remote Broker automatically. This is when we are talking about Outgoing messages.

In the case of Incoming messages, the Local Broker subscribes to the messages sent by publishers with valid user/password combination and the accepted topics (as it does with the ones coming from the LAN). Once the messages are received, they are shared by Mosquitto among all the IoT devices subscribing to these topics in the LAN. In order to receive messages from the internet, this Local Broker is exposed in to the internet in the DMZ of the AP. Each ESP8266 running the Code-1, will receive the message, verify the payload, and that should end with its own IP address. This indicates to the IoT device that it must take this order for it own. When the topic doesn't end with the IP of the device, the message is ignored.

Without the Local Broker, only one ESP8266 would be exposed to the Internet, and it would be responsible of dealing with all the devices deployed in its LAN. Encrypting own and other devices messages with less than half MB RAM is beyond its capabilities.

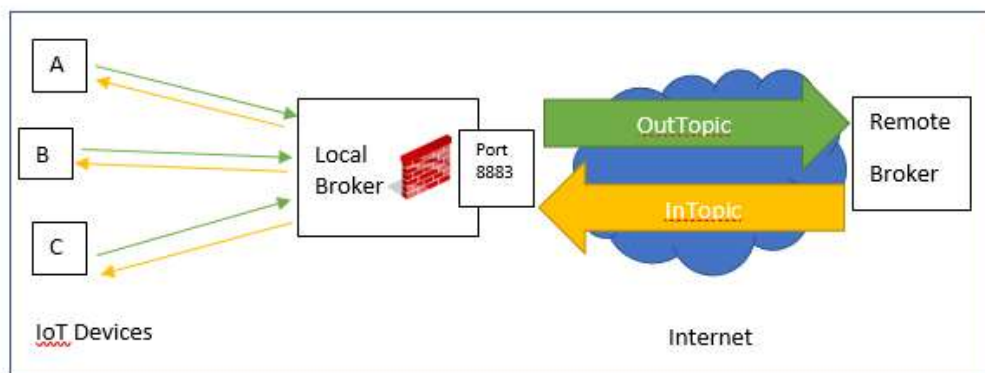


Figure-12 Mosquitto Local Broker.

In the Figure-12, all the IoT devices publish and subscribe topics in the local broker, and they are connected to the same LAN. Meanwhile, the port 8883 of the Local Broker is exposed to the

internet to subscribe incoming topics and to establish the bridge to the Remote Broker. The firewall (provided by the AP), exposes the Local Broker in the demilitarized zone, in consequence the traffic coming to its public IP is going to be referred to the Local Broker.

We must remind that Figure-12 doesn't restrict the source of InTopic to the Remote Broker and any device connected to the Internet could publish topics to the Local Broker. Although, the Local broker subscribes to InTopic, as long as:

- 1- It matches to the subscribed topic.
- 2- The publisher has a valid username/password combination.
- 3- The message format corresponds to anyone of the accepted topic/payloads.
- 4- In the topic structure a valid and active IP address corresponds to any of the connected IoT devices.

If a single one of the conditions if not satisfied, the message will not be considered either by the Local Broker or any of the IoT devices.

LAMP (Linux, Apache, MySQL, PHP) was installed in the local Broker to perform tests, run python applications to publish/subscribe topics, and data storage routines, along with Mosquitto and PAHO (python libraries for Mosquitto).

C-ACCESS POINT



Figure-13 DMZ and Port Forwarding setup Access Point

While 192.168.1.16 (the Local Broker) is exposed to the internet, all the ports but 8883, that are not used for any required function, should be closed to avoid attacks that compromise the security of the Local Broker and/or the LAN. (Figure-13).



Figure-14. Our IoT device highlighted in red.

D-VMWARE ESXI SERVER AND VMWARE VSWITCH

Enabling two interfaces on the Dell PowerEdge 430, one for Management and another one for Internet access, they were configured to connect through vSwitch0 and vSwitch1 (Figure-3). Two networks, VM Network (Management) and VM Internet (Internet).

VM Network IP address is 10.3.32.103. Using Cisco VPN client to establish connection, VMware vSphere Hypervisor gives control of VMware ESXi.

VM Internet connection allows to use one public IP (129.128.116.190). This public IP address is basic to build the MQTT Bridge.

E-OPENSTACK VMWARE VM RUNNING WITH CENTOS SERVER 7.

Once the CENTOS Server 7 ISO was uploaded to VMware ESXi Datastore, it was used to install the OS. Centos Server doesn't install by itself the GUI, and you must configure it after finishing the Centos Server setup. The NIC connected to vSwitch1 will be our source for Internet connection to accomplish the following phases during the setup, and to connect the MQTT Bridge.

As we were provided with a Public IP, we will set up the interface connected to Network "Internet" with the following parameters:

- IPADDR=129.128.116.190
- GATEWAY=129.128.116.161
- MASK=255.255.255.224
- DNS1=129.128.5.233
- DNS2=8.8.8.8
- NM_CONTROLLED=no

The last parameter is critical, because we don't want the OS to change our interface configuration. Having internet connection, we follow with this task list:

1. Update the OS: `yum update -y`
2. Install `wget` and `git` (they will be usefull):
 - a. `yum install wget -y`
 - b. `yum install git -y`
3. Install `mosquitto` and its libraries (although it is not going to be used in this machine, is useful for tests):
 - a. `wget`
`http://download.opensuse.org/repositories/home:/oojah:/mqtt/CentOS_CentOS-5/home:oojah:mqtt.repo`
 - b. `yum update`
 - c. `yum install mosquitto`
 - d. `yum install mosquitto-clients`
4. Install LAMP (we are interested in MySQL and Python).
 - a. `yum install mysql-server`
 - b. `service mysqld start`
 - c. `/usr/bin/mysql_secure_installation`
 - d. `yum install php php-mysql`
5. Install the GUI
 - a. `yum -y groups install "GNOME Desktop"`

The reason for installing a GUI is to have access to the OpenStack Dashboard, that requires a Web Browser and to make easier the installation processes. CentOS (a RedHat based OS), uses “yum” repositories. The selected OpenStack deployment is an AIO (All in One) named PackStack that installs OpenStack-Newton. The setup process is as follows:

Commands to be typed in terminal	Comments
1. <code>sudo systemctl disable firewalld</code> 2. <code>sudo systemctl stop firewalld</code>	Stop and disable the firewall. OpenStack is going to create interfaces and will use them and the already existing to connect the services.
3. <code>sudo systemctl disable NetworkManager</code> 4. <code>sudo systemctl stop NetworkManager</code>	Stop and disable the NetworkManager. The interfaces should be manually set up and, as already said, OpenStack will setup interfaces for its own service.
5. <code>sudo systemctl enable network</code> 6. <code>sudo systemctl start network</code>	Start Network service (in case it was not already enabled and started).
7. <code>sudo yum install -y centos-release-OpenStack-newton</code>	Install the Centos OpenStack Newton repository
8. <code>sudo yum update -y</code>	Update CentOS. This will update all installed apps.
9. <code>sudo yum install -y OpenStack-packstack</code>	Install PackStack.
10. <code>sudo packstack --allinone</code>	Start Packstack AIO

Code-3. OpenStack Installation


```

ACCEPT tcp -- anywhere anywhere multiport dports http /* 001 nagios incoming */
ACCEPT tcp -- localhost.localdomain anywhere multiport dports 5666 /* 001 nagios-nrpe incoming nagios_nrpe */
ACCEPT udp -- anywhere anywhere multiport dports bootps /* 001 neutron dhcp in incoming neutron_dhcp_in_129.128.116.190
*/
ACCEPT tcp -- anywhere anywhere multiport dports 9696 /* 001 neutron server incoming neutron_server_129.128.116.190 */
ACCEPT udp -- localhost.localdomain anywhere multiport dports 4789 /* 001 neutron tunnel port incoming
neutron_tunnel_129.128.116.190_129.128.116.190 */
ACCEPT tcp -- anywhere anywhere multiport dports 8773,8774,8775 /* 001 nova api incoming nova_api */
ACCEPT tcp -- localhost.localdomain anywhere multiport dports rfb:cvsup /* 001 nova compute incoming nova_compute */
ACCEPT tcp -- localhost.localdomain anywhere multiport dports 16509,49152:49215 /* 001 nova qemu migration incoming
nova_qemu_migration_129.128.116.190_129.128.116.190 */
ACCEPT tcp -- anywhere anywhere multiport dports 6080 /* 001 novncproxy incoming */
ACCEPT tcp -- localhost.localdomain anywhere multiport dports 6379 /* 001 redis service incoming redis service from 129.128.116.190
*/
ACCEPT tcp -- anywhere anywhere multiport dports webcache /* 001 swift proxy incoming swift_proxy */
ACCEPT tcp -- localhost.localdomain anywhere multiport dports x11,6001,6002,rsync /* 001 swift storage and rsync incoming
swift_storage_and_rsync_129.128.116.190 */

Chain FORWARD (policy ACCEPT)
target prot opt source destination
neutron-filter-top all -- anywhere anywhere
neutron-openvswi-FORWARD all -- anywhere anywhere
nova-filter-top all -- anywhere anywhere
nova-api-FORWARD all -- anywhere anywhere
ACCEPT all -- anywhere anywhere /* 000 forward in */
ACCEPT all -- anywhere anywhere /* 000 forward out */
ACCEPT tcp -- anywhere 10.0.1.8 tcp dpt:ibm-mqisdp
ACCEPT tcp -- anywhere 10.0.1.8 tcp dpt:secure-mqtt

Chain OUTPUT (policy ACCEPT)
target prot opt source destination
neutron-filter-top all -- anywhere anywhere
neutron-openvswi-OUTPUT all -- anywhere anywhere
nova-filter-top all -- anywhere anywhere
nova-api-OUTPUT all -- anywhere anywhere
ACCEPT udp -- anywhere anywhere multiport dports bootpc /* 001 neutron dhcp out outgoing neutron_dhcp_out_129.128.116.190
*/

Chain neutron-filter-top (2 references)
target prot opt source destination
neutron-openvswi-local all -- anywhere anywhere

Chain neutron-openvswi-FORWARD (1 references)
target prot opt source destination
neutron-openvswi-sg-chain all -- anywhere anywhere PHYSDEV match --physdev-out tapaa483725-00 --physdev-is-bridged /*
Direct traffic from the VM interface to the security group chain. */
neutron-openvswi-sg-chain all -- anywhere anywhere PHYSDEV match --physdev-in tapaa483725-00 --physdev-is-bridged /*
Direct traffic from the VM interface to the security group chain. */

Chain neutron-openvswi-INPUT (1 references)
target prot opt source destination
neutron-openvswi-oaa483725-0 all -- anywhere anywhere PHYSDEV match --physdev-in tapaa483725-00 --physdev-is-bridged
/* Direct incoming traffic from VM to the security group chain. */

Chain neutron-openvswi-OUTPUT (1 references)
target prot opt source destination

Chain neutron-openvswi-iaa483725-0 (1 references)
target prot opt source destination
RETURN all -- anywhere anywhere state RELATED,ESTABLISHED /* Direct packets associated with a known session to the
RETURN chain. */
RETURN udp -- 10.0.1.2 anywhere udp spt:bootps udp dpt:bootpc
RETURN tcp -- anywhere anywhere tcp dpt:secure-mqtt
RETURN icmp -- anywhere anywhere
RETURN tcp -- anywhere anywhere tcp dpt:ssh
RETURN all -- anywhere anywhere match-set NIPv4fa8b34e6-ffcc-4a33-954d- src
RETURN tcp -- anywhere anywhere tcp dpt:ibm-mqisdp
DROP all -- anywhere anywhere state INVALID /* Drop packets that appear related to an existing connection (e.g. TCP
ACK/FIN) but do not have an entry in conntrack. */
neutron-openvswi-sg-fallback all -- anywhere anywhere /* Send unmatched traffic to the fallback chain. */

```

Chain neutron-openvswi-local (1 references)				
target	prot	opt	source	destination
Chain neutron-openvswi-oaa483725-0 (2 references)				
target	prot	opt	source	destination
RETURN	udp	--	default	255.255.255.255
neutron-openvswi-saa483725-0	all	--	anywhere	anywhere
RETURN	udp	--	anywhere	anywhere
DROP	udp	--	anywhere	anywhere
RETURN	all	--	anywhere	anywhere
RETURN	chain.	/*		
RETURN	all	--	anywhere	anywhere
DROP	all	--	anywhere	anywhere
ACK/FIN)	/* Drop packets that appear related to an existing connection (e.g. TCP			
neutron-openvswi-sg-fallback	all	--	anywhere	anywhere
/* Send unmatched traffic to the fallback chain. */				
Chain neutron-openvswi-saa483725-0 (1 references)				
target	prot	opt	source	destination
RETURN	all	--	10.0.1.8	anywhere
DROP	all	--	anywhere	anywhere
/* Drop traffic without an IP/MAC allow rule. */				
Chain neutron-openvswi-sg-chain (2 references)				
target	prot	opt	source	destination
neutron-openvswi-iaa483725-0	all	--	anywhere	anywhere
/* Jump to the VM specific chain. */				
neutron-openvswi-oaa483725-0	all	--	anywhere	anywhere
/* Jump to the VM specific chain. */				
ACCEPT	all	--	anywhere	anywhere
Chain neutron-openvswi-sg-fallback (2 references)				
target	prot	opt	source	destination
DROP	all	--	anywhere	anywhere
/* Default drop rule for unmatched traffic. */				
Chain nova-api-FORWARD (1 references)				
target	prot	opt	source	destination
Chain nova-api-INPUT (1 references)				
target	prot	opt	source	destination
ACCEPT	tcp	--	anywhere	localhost.localdomain tcp dpt:8775
Chain nova-api-OUTPUT (1 references)				
target	prot	opt	source	destination
Chain nova-api-local (1 references)				
target	prot	opt	source	destination
Chain nova-filter-top (2 references)				
target	prot	opt	source	destination
nova-api-local	all	--	anywhere	anywhere

TABLE-4. IPTABLES including OpenStack rules.

One last procedure related to the IPTABLES is pending. It is sharing the Internet connection by executing the following commands:

- *service iptables*
- *iptables -t nat -A POSTROUTING -o ens32 -j MASQUERADE*
- *service iptables save*
- *service iptables restart*

On Table-4, highlighted in yellow, you can see the entries to forward MQTT traffic to the 10.0.1.8, the IP of the OpenStack Instance running Mosquitto.

```

[root@localhost ~]# ip route
default via 129.128.116.161 dev ens32
10.0.1.0/24 via 172.24.4.228 dev br-ex
129.128.116.160/27 dev ens32 proto kernel scope link src 129.128.116.190
169.254.0.0/16 dev ens32 scope link metric 1002
169.254.0.0/16 dev br-ex scope link metric 1007
172.24.4.224/28 dev br-ex proto kernel scope link src 172.24.4.225
[root@localhost ~]#

```

Figure-15 Ip route CentOS VM.

Figure-15 shows two key routes, The first one is the default route via “ens32” for all traffic without any specific route, and the second one that tells CentOS that all traffic for 10.0.1.0/24 (the Internal OpenStack Network) should go to 172.24.4.228 (The OpenStack Router Interface connected to br-ex) via br-ex interface.

When PackStack finishes the installation and booting all OpenStack Services, it creates `keystonerc_admin` file, that is useful for authentication. One way to use it is typing the command:

source keystonerc_admin

This will use the admin password that is stored in this file and will authenticate the admin user in OpenStack. Now is possible to work with OpenStack from the Terminal console. For example we can obtain the list of OpenStack enabled services:

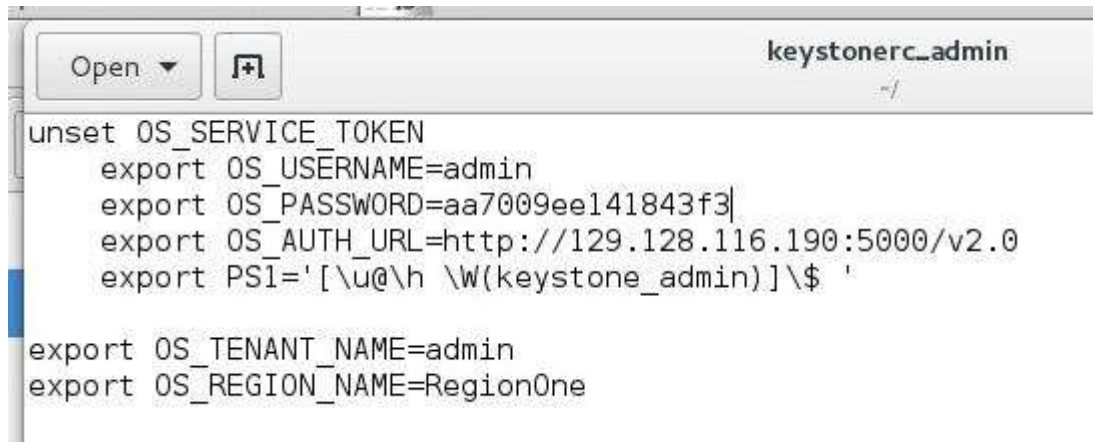
```

[root@localhost ~(keystone_admin)]# openstack service list
+-----+-----+-----+
| ID                    | Name      | Type      |
+-----+-----+-----+
| 0ed9483c1ee844699abf50109be5e2fb | glance   | image     |
| 20eaadd64f884b2fa89869d63f34e365 | ceilometer | metering  |
| 28ccb61d2828412cb1480bdabf279f29 | cinderv2  | volumev2  |
| 3a55ccd7052c4ec3b389e6dc4a3bb421 | cinder    | volume    |
| 519a71a8386c4b3db843574efb5cc1ff | swift     | object-store |
| 5e3859ed9832435eac44f1303a3c292f | keystone  | identity  |
| 6427a921b4274f7bbe2ff786dc451b1a | aodh      | alarming  |
| cd029e55870846099a8d7f763f4b1cf9 | neutron   | network   |
| d7a6ced5629844d68f1b57218886295c | cinderv3  | volumev3  |
| ed214b9a3c5f47d3b3fd9eac03c21505 | gnocchi   | metric    |
| f5b77b20f1ad4b3b9c641640379f982b | nova      | compute   |
+-----+-----+-----+
[root@localhost ~(keystone_admin)]# █

```

Figure-16 OpenStack Services

Also, you can use the admin password stored in this file (`keystonerc_admin`) to open the Dashboard in a web browser with admin user. (Figure-17)

A terminal window titled 'keystonerc_admin' with a standard Linux shell prompt. It contains several export and unset commands for configuring the OpenStack Keystone service. The commands are: 'unset OS_SERVICE_TOKEN', 'export OS_USERNAME=admin', 'export OS_PASSWORD=aa7009ee141843f3', 'export OS_AUTH_URL=http://129.128.116.190:5000/v2.0', 'export PS1='\u@\h \W(keystone_admin)]\\$ ', 'export OS_TENANT_NAME=admin', and 'export OS_REGION_NAME=RegionOne'.

```
unset OS_SERVICE_TOKEN
export OS_USERNAME=admin
export OS_PASSWORD=aa7009ee141843f3
export OS_AUTH_URL=http://129.128.116.190:5000/v2.0
export PS1='\u@\h \W(keystone_admin)]\$ '

export OS_TENANT_NAME=admin
export OS_REGION_NAME=RegionOne
```

Figure-17. OpenStack admin password (Keystone service)

F-USER PC AND/OR SMARTPHONE:

Optional, but not less important, an “UserPC” and/or “Smartphone” could be used to interact with the system, either to modify the IoT Sensors setup, or to retrieve data from the Database Server.

G-OPENSTACK NETWORKS, ROUTER, IMAGES AND INSTANCE.

OpenStack is a cloud virtualization tool that provides many useful solutions. For simplification purposes, we will focus on Networks, Router, Images and Instance setup. The Dashboard allows you to setup virtualized devices in OpenStack. To load the Dashboard (called Horizon in other versions) you can do it from the web, accessing entering the following ip address in the web browser address bar:

- “129.128.116.190/dashboard” from remote location using Internet
- “localhost/dashboard” from the CentOS VM web browser.

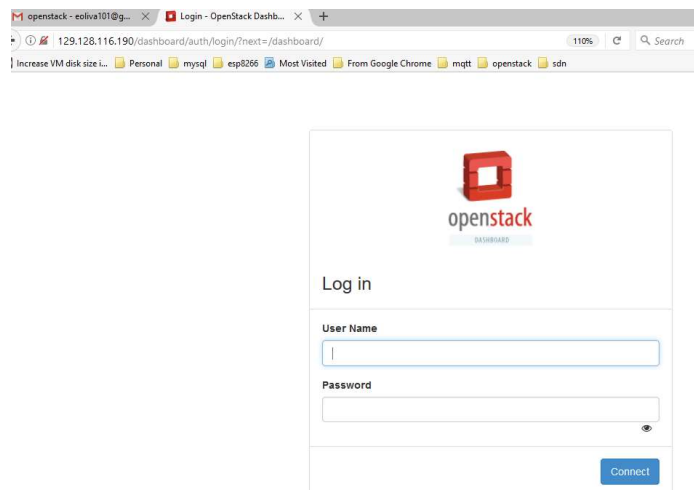


Figure-18. OpenStack Dashboard Log-In

User Name= admin
Password= aa7009ee14183f3

After you are logged-in, you can start creating your own project or proceed to build a project inside the already created “admin” project. Following, the list of steps to have an Instance running in OpenStack:

1)Create Host aggregates: Clic Admin/System/Host Aggregates, to create the Internal Availability zone and the GasD Host Aggregate that will be useful for performance assessment and billing data consolidation.

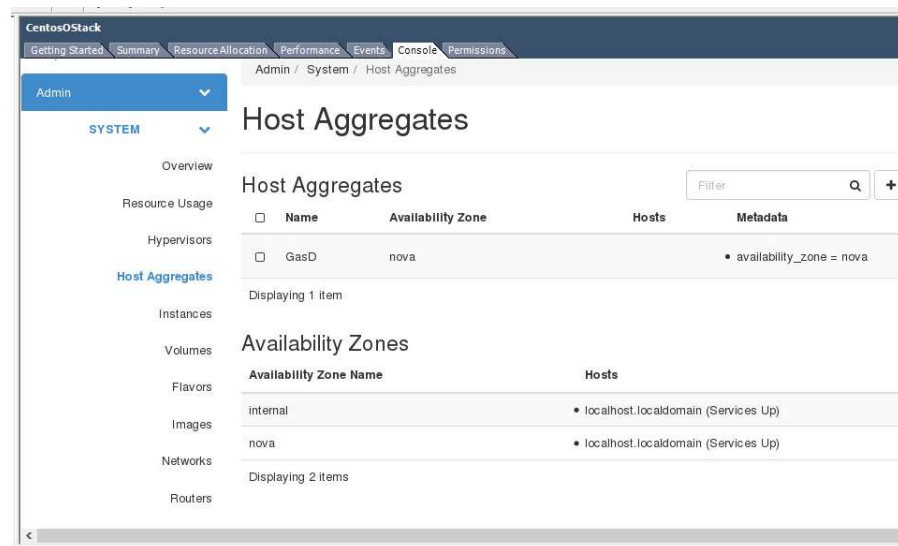


Figure-19. OpenStack Host Aggregates

2)Create the Internal Network that will be used to connect the Instance where our application and database server are going to be working. The Public database was created by Packstack during the setup. Click Project/Network/Networks and create the Internal network. Figure

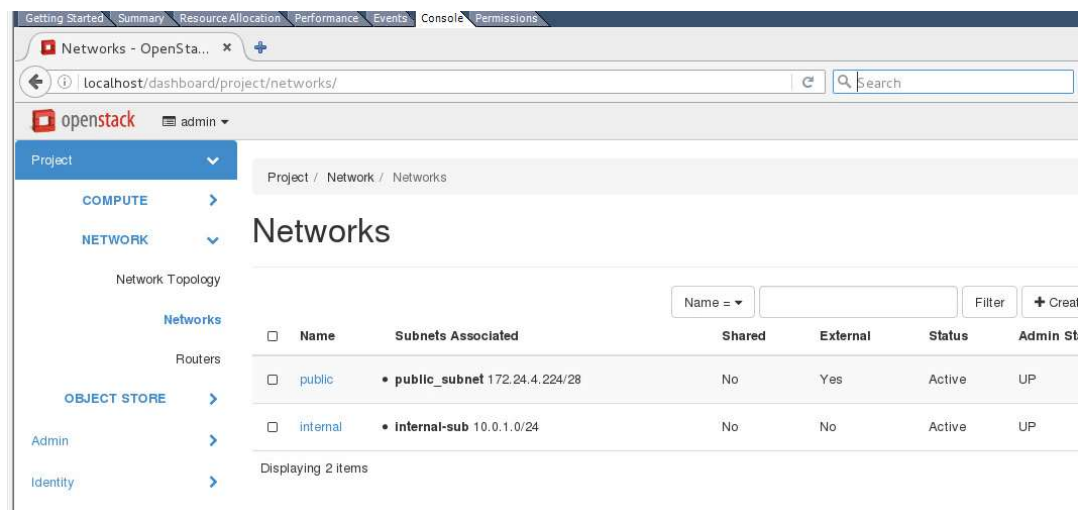


Figure-20. OpenStack Public and Internal Network

3) Once the Networks are ready, it is time to create the Router, that using Static Routes will forward the packets between Internal and Public networks

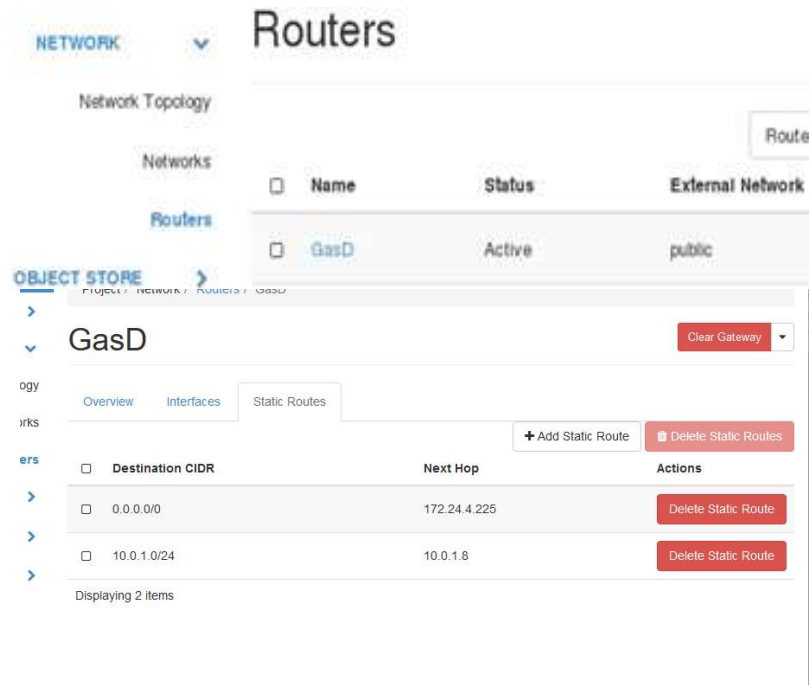


Figure-21. OpenStack Router and Static Routes

4) OpenStack uses the concept of Images and Flavors. The Flavors are pre-made templates to create Instances. They specify features like number of processors, RAM and Volume size. If there is not any Flavor matching your needs, you can edit them before building and Instance. In our case I edited **m1.large** to be able to create an instance with 4vCPUs, 20GB Drive and 8GB RAM.

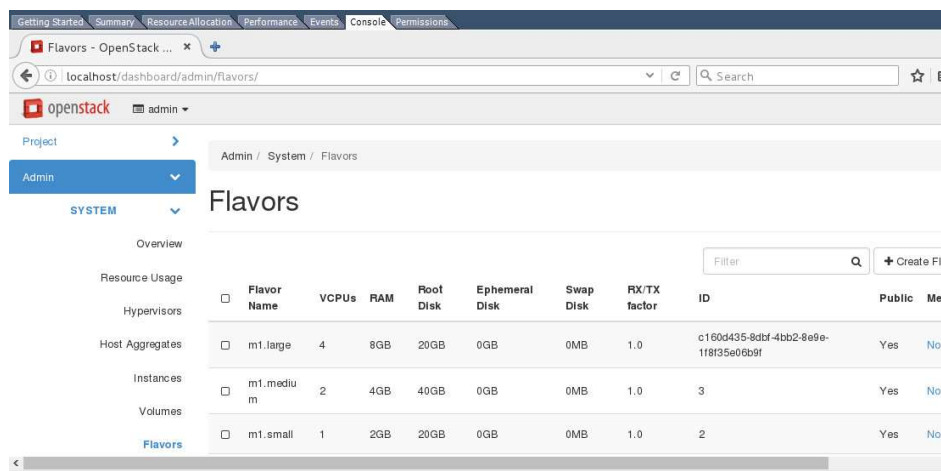


Figure-22. OpenStack Flavors.

The images are used to create Instances based on Volume Images that can be based in certain Operating System IOS or a previously created volume. OpenStack can upload a file from the Host Machine, to build an Image that will be used to install the Instance.

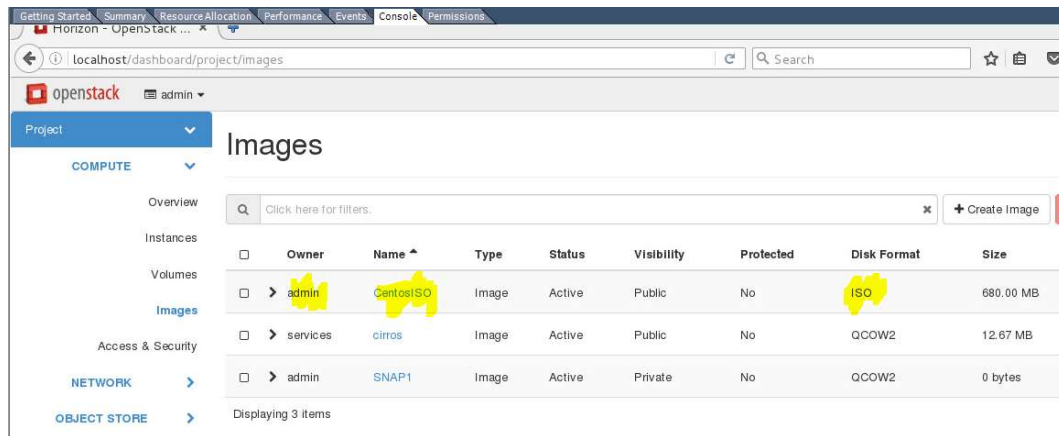


Figure-23. OpenStack Image CentosISO loaded with CentOS Server 7 ISO.

5)The Instance, Virtual Machine in OpenStack, can be created based in a flavor and using an Image loaded with the Installation Package of an OS. During its creation you must provide:

- The Name.
- The Image used to boot (Source).
- The Flavor it is based in.
- The Network is connected to (Internal), this will result in the assigned IP (10.0.1.8).
- The KeyPair (GasKey)Optional to have ssh access to the cloud.

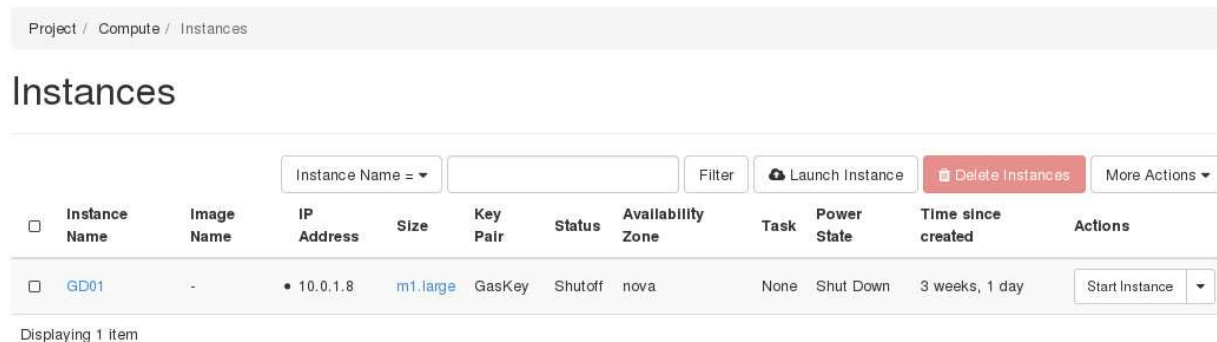


Figure-24. OpenStack Instance GD01.

If you have enough resources to build this instance, it will be created and will Start by itself. But so far it doesn't have a volume. Previously you could create a Volume and leave it ready to be attached to the Instance (despite it is already working). The DropDown button under Actions, has a specific action to attach a volume. Meanwhile, the Instance is loading CentOS installation. At this point you have a working OpenStack Instance.

H-REMOTE BROKER

Having an OpenStack cloud running an Instance takes us closer to accomplishing the Remote Broker. Once CentOS is installed, the following steps are:

1-Login into the GD01 Terminal. Two ways are available:

- a) Click GD01 name and open the console.
- b) From any system connected to internet, “ssh 129.128.116.190” and after “ssh 10.0.1.8”. This procedure is less resources demanding and provides faster CLI response.

```
login as: root
root@129.128.116.190's password:
Access denied
root@129.128.116.190's password:
Last failed login: Tue Feb  7 22:59:29 EST 2017 from d75-158-127-185.abhsia.telus.net on ssh:notty
There were 23 failed login attempts since the last successful login.
Last login: Tue Feb  7 22:54:42 2017 from 129.128.116.162
[root@localhost ~]# ssh 10.0.1.8
root@10.0.1.8's password:
Last failed login: Tue Feb  7 22:56:23 EST 2017 from 172.24.4.225 on ssh:notty
There were 2 failed login attempts since the last successful login.
Last login: Tue Feb  7 22:52:11 2017
[root@host-10-0-1-8 ~]#
```

Figure-25. SSH VM hosting Cloud and Remote Broker.

2-Install the utilities and apps that we need to set up our broker. This Instance is getting an IP from OpenStack (you can verify in the Instances tab). We will set up this interface connected to the Internal Network of OpenStack with the following parameters:

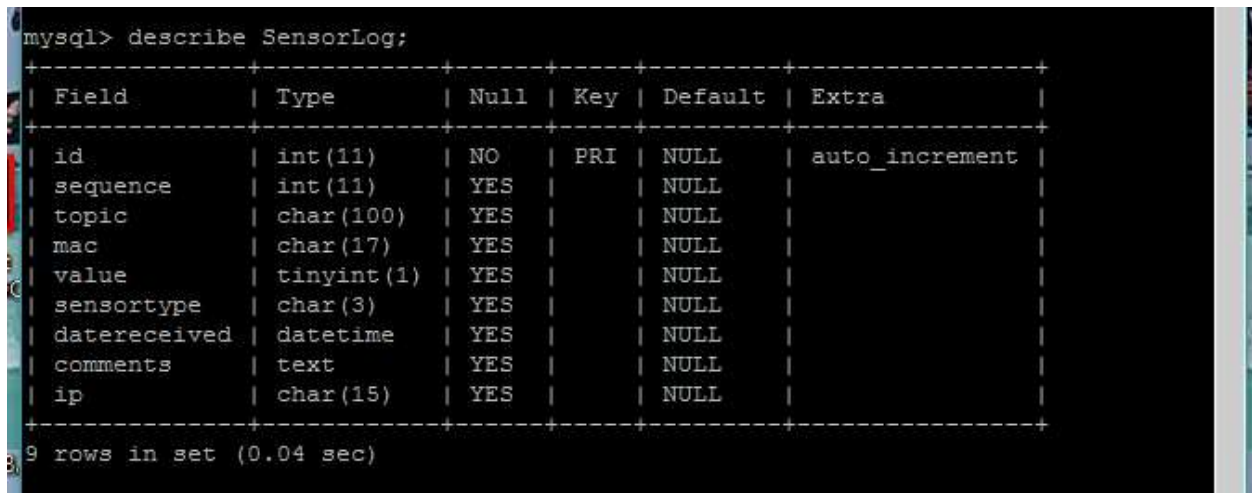
- IPADDR=10.0.1.8
- GATEWAY=10.0.0.1
- MASK=255.255.255.0
- DNS1=129.128.5.233
- DNS2=8.8.8.8
- NM_CONTROLLED=no

The last parameter is critical, because we don't want the OS to change our interface configuration. Having internet connection, we follow with this task list:

1. Update the OS: yum update -y
2. Install wget and git (they will be usefull):
 - a. yum install wget -y
 - b. yum install git -y
3. Install mosquitto and its libraries (although it is not going to be used in this machine, is useful for tests):
 - a. wget
http://download.opensuse.org/repositories/home:/oojah:/mqtt/CentOS_CentOS-5/home:oojah:/mqtt.repo. Or, we can use “scp root@172.24.4.225:/<path to repo folder>/* ./ to copy from the previous installation process.
 - b. yum update
 - c. yum install mosquitto
 - d. yum install mosquitto-clients

4. Install LAMP (we are interested in MySQL and Python).
 - a. yum install mysql-server
 - b. service mysqld start
 - c. /usr/bin/mysql_secure_installation
 - d. yum install php php-mysql

3-In MySQL a Database “GasDetector” was created with a table SensorLog, with the following structure:



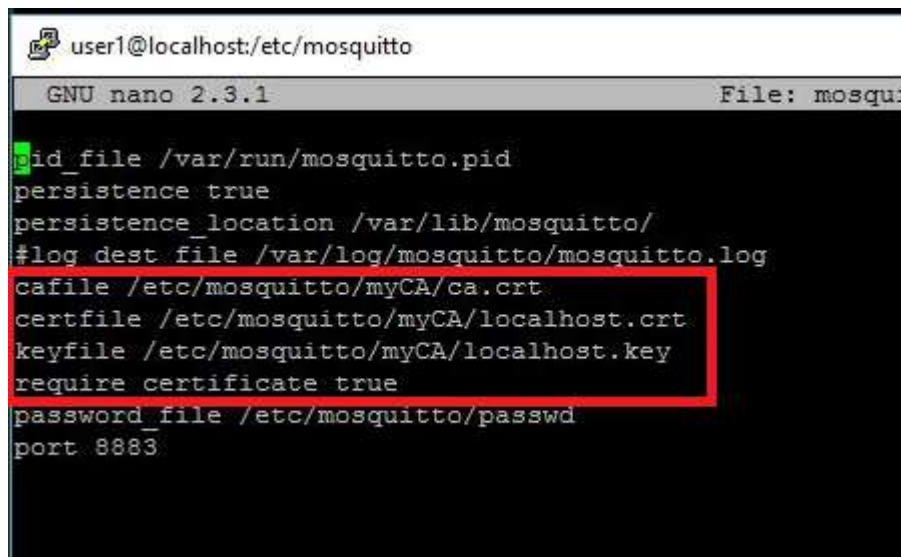
```
mysql> describe SensorLog;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
sequence	int(11)	YES		NULL	
topic	char(100)	YES		NULL	
mac	char(17)	YES		NULL	
value	tinyint(1)	YES		NULL	
sensortype	char(3)	YES		NULL	
datereceived	datetime	YES		NULL	
comments	text	YES		NULL	
ip	char(15)	YES		NULL	

9 rows in set (0.04 sec)

Figure-26. gasDetector database, SensorLog Table structure in MySql;

2-Install Mosquitto. Mosquitto libraries, Publisher and Subscriber. Also we must create the password file (Passwd).



```
user1@localhost:/etc/mosquitto
GNU nano 2.3.1 File: mosquitto.conf

pid_file /var/run/mosquitto.pid
persistence true
persistence_location /var/lib/mosquitto/
#log dest file /var/log/mosquitto/mosquitto.log
cafile /etc/mosquitto/myCA/ca.crt
certfile /etc/mosquitto/myCA/localhost.crt
keyfile /etc/mosquitto/myCA/localhost.key
require_certificate true
password_file /etc/mosquitto/passwd
port 8883
```

Figure-27. Mosquitto.conf file

With this configuration Mosquitto will be ready to accept publishers using the secured port 8883 and will require UserID and Password as well. By now, will be better to comment out (# in the first line character), the lines inside the red box, because the will be used in a separate section.

3-Install PAHO and Mysql libraries for Python. Develop the subscriber code (GasDetector.py). This application will subscribe to the topic of interest and will save data coming in the payload that matches with the requires structure. Is it doesn't match, then an exception will be caused and the message will be discarded.

GasDetector.py	Comments
<pre> import paho.mqtt.client as mqtt import mysql from time import gmtime, strftime from mysql.connector import errorcode import sys import MySQLdb userID="user1" password="silvana01" def storedata(topic,info): # you must create a Cursor object. It will let # you execute all the queries you need cur = dbase.cursor() try : i=0 l=list(info) j=0 if l[i]=='R': comments="Requested reading" i=i+10 j=i else: comments="Regular reading" while l[i]!='.': i=i+1 sequence=long("".join(l[j:i])) print (sequence) i=i+1 j=i while l[i]!=' ': i=i+1 sensortype="".join(l[j:i]) print (sensortype) i=i+11 j=i while l[i]!=' ': i=i+1 ipv4="".join(l[j:i]) print (ipv4) i=i+5 j=i while l[i]!=' ': i=i+1 mac="".join(l[j:i]) print (mac) i=i+15 vvalue="".join(l[i:i+1]) if vvalue=="F": vvalue="0" if vvalue=="T": vvalue="1" print (vvalue) ttime=strftime("%Y-%m-%d %H:%M:%S", gmtime()) print (ttime) sql="insert into SensorLog (sequence,topic,ip,mac,value,sensortype,datereceived,comments) values (%s,%s,%s,%s,%s,%s,%s,%s) " </pre>	<p>Import all required libraries.</p> <p>MQTT user and password</p> <p>Function to store message containing message with right structure.</p> <p>Extract fields from message.</p> <p>Build SQL instruction</p>

<pre> args= (sequence,topic,ipv4,mac,int(vvalue),sensortype,time,comments) cur.execute(sql,args) dbase.commit() print("data was stored succesfully") except: print("Unexpected error:", sys.exc_info()[0]) def on_message(client, userdata, msg): print("TOPIC:" + msg.topic + " payload:" + str(msg.payload)) storedata(msg.topic,msg.payload) def on_connect(client, userdata, flags, rc): print("Connected with result code "+str(rc)) client.subscribe("outTopic/#") try: dbase = MySQLdb.connect(host="localhost", # your host, usually local\$ user="root", # your username passwd="silvana01", # your password db="gasDetector") # name of the data base except mysql.connector.Error as err: if err.erno == errorcode.ER_ACCESS_DENIED_ERROR: print("Something is wrong with your user name or password") elif err.erno == errorcode.ER_BAD_DB_ERROR: print("Database does not exist") else: print(err) sys.exit() client = mqtt.Client() client.on_connect = on_connect client.on_message = on_message client.username_pw_set(userID, password) client.connect("localhost", 8883) try: client.loop_forever() except (KeyboardInterrupt, SystemExit): dbase.close raise </pre>	<p>Prepare Arguments to pass to SQL Commit changes to database</p> <p>If exception is captured then report on message.</p> <p>Create MQTT connection function Subscribe to topic</p> <p>Connect to Database</p> <p>Capture error connecting to database</p> <p>Create MQTT client</p> <p>Validate client user and pwd</p> <p>Capture KB exception Close database on exception and exit.</p>
--	---

Code-4. Subscriber application, GasDetector.py

I-SETUP SUMMARY and PASSWORDS

After going through the entire Setup process in this section, we should have:

- A WGD connected to the LAN of the facility where it is going to be deployed, and publishing topics in a Local Broker.
- A Local Broker bridge-connected to a Remote Broker, and resending all the topics that are declared in the bridge. Also this broker is able to subscribe topics from publishers (like a mobile app or other PC connected to the internet) trying to update the WGD configuration. The enabled update are: 1)Change the frequency of readings, 2)Request a reading in this precise moment), but more parameters can be adjusted with the same procedure.
- An OpenStack cloud running an Instance for Data Processing and Database Management.
- A MQTT Remote Broker subscribing topics from Local Routers.

In consequence, some usernames and passwords were created to secure the system:

System/Device	Username	Password
Local MQTT Broker	user1	silvana01
Lab's Server (IP 10.3.32.103)	root	sdnlabs#103
Ubuntu 14.04 (Local Broker OS)	root	31052005scor

OpenStack Cloud Dashboard on Public IP 129.128.116.190	admin	aa7009ee14183f3
CentOS Stack VMware VM	root	aR7576196-R
CentOS OpenStack Instance	root	yV_353680257Union
Remote MQTT Broker	user1	silvana01
MySQL	root	silvana01

TABLE-5. UserNames and Passwords.

THE SECURITY

The potential of the IoT is only comparable to its intrinsic weakness. The only way to achieve fast performance with low power consumption is making focus on the basic functions and leaving on a side all that can be delegated to systems with more resources (cpu-memory-storage) and power supply (AC).

The integrated low power 32-bit MCU (in ESP8266-01) can has very interesting features, but SSL is not included in the list. Then, transmitting from a site location to a server using internet, will create a large route where possible sniffers could easily steal the UserID and the Password. Assuming that the LAN is well secured and properly set up, then installing our “own man in the middle” could help to achieve a higher level of security.²² Having an extra component or intermediate device could be seen as a redundant or additional cost, but when handling information like “hazardous gases presence”, we can’t open the door to third parties becoming the strange man in the middle. Mosquitto is TLS (Transport Layer Security) capable and IANA designed port 8883 port for its implementation.

The Local Broker is this “nice-man” in the middle and has three security duties:

- 1) Encrypt all the messages that are going through Internet on their way to the Remote Broker.
- 2) Validate all the incoming topics before passing them to the WGSs.
- 3) Hide our “by design-low security” devices from possible attacks.

Among all the MQTT-SSL posts available on the Internet, the “generate-CA.sh” script²³ is very useful and is the most popular to generate self-signed certificates (for the Certification Authority, the Server and Client).

In this occasion, we created:

- One certificate authority.
 - ca.crt, ca.srl, ca.key files.
- One server certificate.
 - localhost.crt, localhost.csr, localhost.key files
- And two clients
 - “client1” for the bridge
 - “paho” for the program GasDetector.py
 - Two set of .crt, .csr and .key files.

Once the certificates and keys were generated, it is time to perform some modifications to our setup:

- a. Local Broker mosquitto.conf file.
- b. Remote Broker mosquitto.conf file.
- c. Add the Intermediate Broker.
- d. GasDetector.py code.

²² <https://www.justinribeiro.com/chronicle/2012/11/08/securing-mqtt-communication-between-arduino-and-mosquitto/>

²³ <https://github.com/owntracks/tools/raw/master/TLS/generate-CA.sh>

File to Update	New Configuration lines	Comments
Local Broker mosquitto.conf	... bridge BridgeIt bridge_cafile /etc/mosquitto/myCA/ca.crt bridge_certfile /etc/mosquitto/myCA/client1.crt bridge_keyfile /etc/mosquitto/myCA/client1.key address 129.128.116.190:8883 ...	CA authority. Server certificate Client key
Remote Intermediate mosquitto.conf	... cafile /etc/mosquitto/myCA/ca.crt certfile /etc/mosquitto/myCA/localhost.crt keyfile /etc/mosquitto/myCA/localhost.key require_certificate true ... bridge BridgeIt bridge_cafile /etc/mosquitto/myCA/ca.crt bridge_certfile /etc/mosquitto/myCA/client1.crt bridge_keyfile /etc/mosquitto/myCA/client1.key address 10.0.1.8:8883 ...	CA authority. Server certificate Server key Require certs for authentication Bridge to our only Remote Broker And we could have more bridges, one for each Remote Broker in our LAN.
Remote Broker mosquitto.conf	... cafile /etc/mosquitto/myCA/ca.crt certfile /etc/mosquitto/myCA/localhost.crt keyfile /etc/mosquitto/myCA/localhost.key require_certificate true ...	CA authority. Server certificate Server key Require certs for authentication
GasDetector.py	In the main section of the code, where the client connection is established, right after: "client.username_pw_set(userID, password)" add: client.tls_insecure_set(True) client.tls_set("/etc/mosquitto/myCA/ca.crt", "/etc/mosquitto/myCA/paho.crt", "/etc/mosquitto/myCA/paho.key")	Don't verify server's name. Certificate authority Paho client certificate. Paho client key.

TABLE-6. Enable TLS features to the system.

One simple, clear and dangerous vulnerability is left open. Publishing topics from internet on the Local Broker with only Username and Password is the perfect backdoor. The system administrator must assess the trade-off resulting of blocking or leaving this open. The safest option is that all the messages that can affect or modify the WGD performance should be generated inside the LAN.

INTERMEDIATE BRIDGE

With the non-secured connection, a bridge was created and it was enough to forward all incoming messages to port 8883 at the Public IP interface, to the Internal IP of our OpenStack instance (Figure-26). But TLS creates new challenges and a secured connection can't be created with the same configuration.

The SSL Handshaking process fails in our configuration, and this makes impossible to authenticate the certificates. In consequence, the communication process can't go beyond the request and is

timed-out. A simple solution to this problem is creating what we will call “Intermediate Broker”. Therefore, instead of forwarding all incoming communications on port 8883, we can install a Mosquitto Broker in the in the VM with the IP exposed to internet. Then, create another bridge with our Remote Broker. A full secured connection is created from the Local Broker to the Remote Broker thanks to the “Intermediate Broker”.

The Intermediate Broker configuration is similar to the Remote Broker’s, and requires the CA authority files to create its own server and client certificates (using the generate-CA.sh script).

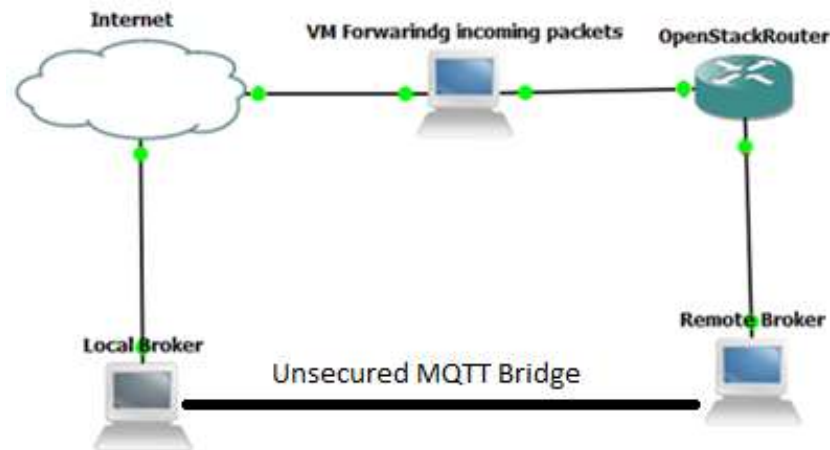


Figure-28. Unsecured MQTT Bridge.

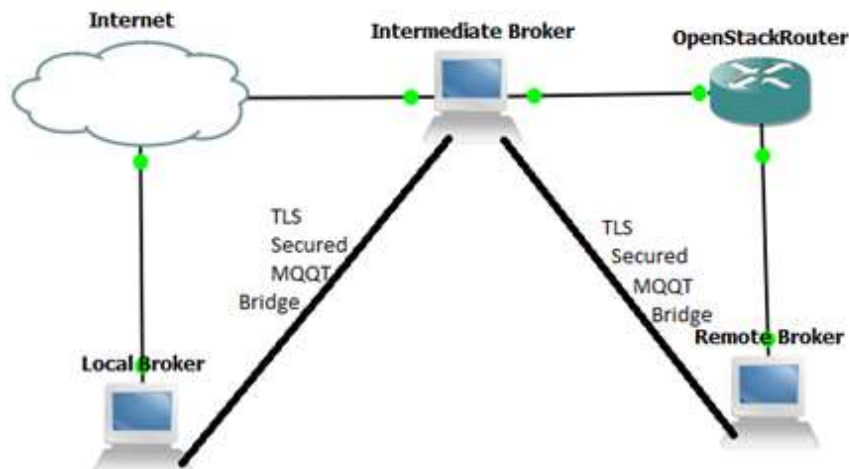


Figure-29. Secured MQTT Bridge.

After installing TLS authentication we have a more secured system, where Brokers are authenticated by username-password combination plus certificates. Also, the communications are done by secured channels and using encryption methods. Some processing load could be saved once the Intermediate Broker is reached, and using unsecured communications inside the Remote Broker’s LAN seems to be reasonable.

Although, encrypting the MQTT messages inside the LAN gives additional protection against breaches in the Intermediate Broker’s security. Also, because we could be using a shared set of

hardware and virtual resources running other applications, encrypting our messages protects them from vulnerabilities of other processes using the same resources.

The Intermediate Broker could be used to distribute topics among many Remote Brokers running in different OpenStack Instances. For example (Figure-29), the Intermediate broker could have bridges to each Remote Broker, and for each one a different set of topics will be delivered. The Intermediate Broker can easily segregate topics in a Cloud for different Mosquitto services or customers that don't want to share computing or storage resources with each other.

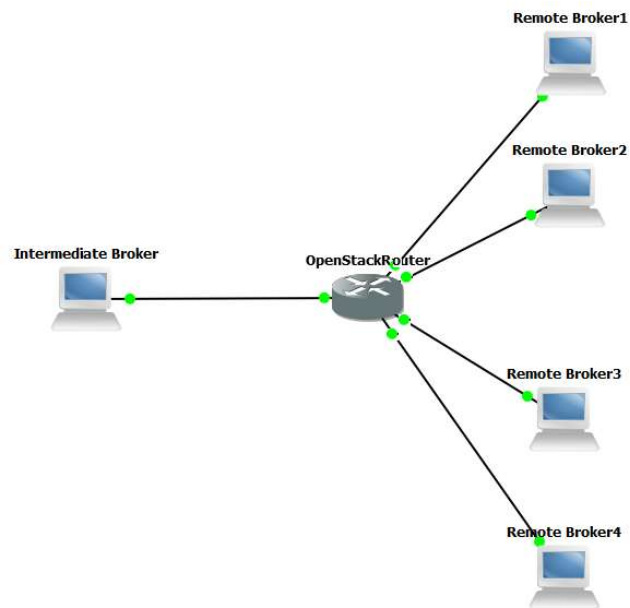


Figure-30. Intermediate Broker.

The `mosquitto.conf` file for the Intermediate Broker (and many bridged Remote Brokers) should include the following settings:

```
cafile /pathtocertificates/ca.crt
certfile /pathtocertificates/server.crt
keyfile /pathtocertificates/server.key

bridge Broker1
remote_cafile /pathtocertificates/ca1.crt
remote_certfile /pathtocertificates/client11.crt
remote_keyfile /pathtocertificates/client11.key
remote_username user1
remote_password password1
address 10.0.1.8:8883
topic outTopic/broker1 both

bridge Broker2
remote_cafile /pathtocertificates/ca2.crt
remote_certfile /pathtocertificates/client21.crt
remote_keyfile /pathtocertificates/client21.key
remote_username user2
```

```
remote_password password2
address 10.0.1.9:8883
topic outTopic/broker2 both
.....
```

Code-5. Intermediate Broker mosquitto.conf file.

Where each *Bridgen* to *Brokern* has a different user-password combination, Certificate Authority and keys as well.²⁴ Before the implementation of the Intermediate Broker, we were only able to forward all traffic on port 8883 to one IP address.

²⁴ Once the Intermediate broker is set-up, the Iptables Forward rule to 10.0.1.8:8883 must be removed.

POWER-ON

It is time to power-on the system and verify it is fully working. A set of steps have to be followed to have all the devices and applications running as they were intended.

A-SERVER/INTERMEDIATE MOSQUITTO BROKER

The first step is to power on our server (located in the lab). Using VMWare Vsphere Client we will log on in the VMWare Hypervisor, If this is done from a remote location (e.g. Internet connection), a Cisco VPN-Client is required to create a tunnel.

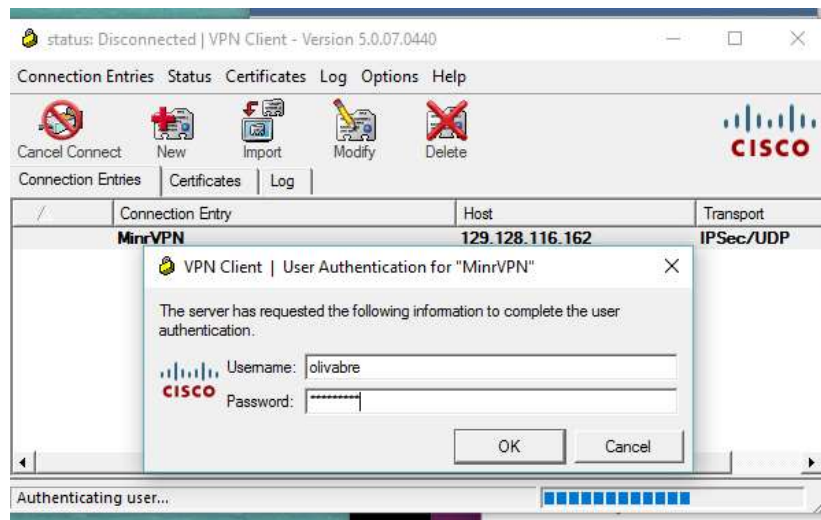


Figure-31. VPN Client



Figure-32. VMWare vSphere Client.

For this, you need the vSphereClient UserName and Password (plus the VPN Client username+password if required). “The Getting Started” tab will receive you and there you can right click your VM in the list of available VMs.

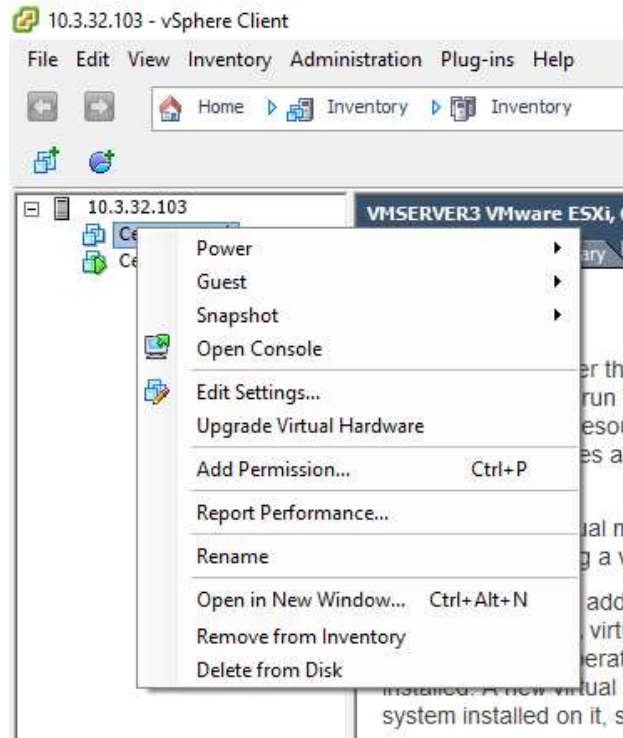


Figure-33 Power-On your Virtual Machine.

The last part related to the server Power-On is to login into CentOS (root user and its password = aR7576196-R are required) and load the GUI. Startx will load CentOS GUI (Centos GNOME Desktop).

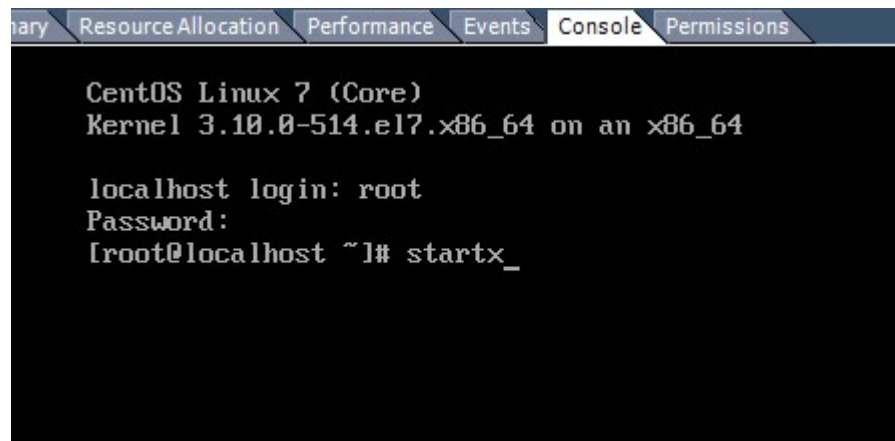


Figure-34. Log in and load GUI.

Having control of the CentOS terminal, is time to start the MQTT Intermediate Broker. The following command will start it in detached mode:

```
mosquitto -d -c /etc/mosquitto/mosquitto.conf
```

The Intermediate Broker will wait for incoming connection requests, and the first request to be received should be the one coming from the Local Broker bridge. Meanwhile the Intermediate Broker is trying to be authenticated by the Remote Broker to create the required bridge. The connection request will be timed out a couple of times until the Remote Broker is up and accepts the connection request.

B-CENTOS INSTANCE/MOSQUITTO REMOTE BROKER

Once the GUI is up, is time to start the dashboard and boot the Instance (using Mozilla Firefox web browser). Figures 15 & 16 show the admin password and the graphic interface of OpenStack. Continue with Figure 22. To have a working instance all you need is to click the “Start Instance” button and it will start up. As described in the Setup section, it is better to access the Instance terminal with SSH, rather than using OpenStack console.

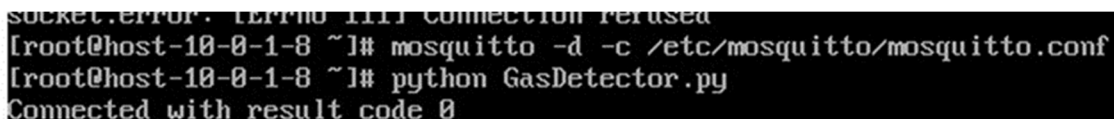
In the command prompt (of the console or a SSH terminal), we must enter the commands to load the broker and the app that will store all the subscribed and valid messages:

```
mosquitto -d -c /etc/mosquitto/mosquitto.conf
```

As result the mosquitto Remote Broker is up (in detached mode using this specific configuration file) and will enable TLS communications on port 8883.

```
python GasDetector.py
```

command will start our application and it will create a connection with mosquitto.



```
socket.error: [Errno 111] Connection refused
[root@host-10-0-1-8 ~]# mosquitto -d -c /etc/mosquitto/mosquitto.conf
[root@host-10-0-1-8 ~]# python GasDetector.py
Connected with result code 0
```

Figure 33. Mosquitto and GasDetector running.

At this point, we will leave the Remote Broker running upon the arrival of messages from our IoT devices.

C-THE LOCAL BROKER

We already installed the Local Broker on a commodity PC, or on a low cost compact system like Raspberry Pi. We must ensure that mosquitto is running and that it is able to connect to the Remote Broker. In this case we will run run mosquitto in verbose mode to see the interaction with the Remote Broker and the IoT devices.

```

dged-T-6859u:/etc/mosquitto$ mosquitto -v -c mosquitto.conf
Warning: Bridge BridgeIt using insecure mode.
487266322: mosquitto version 1.4.10 (build date Thu, 25 Aug 2016 15:32:03 +0100)
starting
487266322: Config loaded from mosquitto.conf.
487266322: Opening ipv4 listen socket on port 8883.
487266322: Opening ipv6 listen socket on port 8883.
487266322: Bridge local.ed-T-6859u.BridgeIt doing local SUBSCRIBE on topic out
pic/
487266322: Bridge local.ed-T-6859u.BridgeIt doing local SUBSCRIBE on topic InT
ic/
487266322: Connecting bridge BridgeIt (129.128.116.190:8883)
487266322: Bridge ed-T-6859u.BridgeIt sending CONNECT
487266322: Received CONNACK on connection local.ed-T-6859u.BridgeIt.
487266322: Bridge local.ed-T-6859u.BridgeIt sending SUBSCRIBE (Mid: 2, Topic:
tTopic/, QoS: 0)
487266322: Bridge local.ed-T-6859u.BridgeIt sending SUBSCRIBE (Mid: 3, Topic:
Topic/, QoS: 0)
487266322: Received PUBACK from local.ed-T-6859u.BridgeIt (Mid: 1)
487266322: Received SUBACK from local.ed-T-6859u.BridgeIt
487266322: Received SUBACK from local.ed-T-6859u.BridgeIt
487266371: New connection from 127.0.0.1 on port 8883.
487266371: New client connected from 127.0.0.1 as mosqpub/28063-ed-T-6859 (c1,

```

Figure-34. Local Broker Bridge requests connection, it is authenticated and working.

Figure 34 is a nice example of Mosquitto in verbose mode. Initially opens the listening port 8883 for IPV4 and IPV6. Then tries to create the bridge with the Intermediate Broker, and once it succeeds it receives a CONNACK message for the bridged connection.. The following lines report that it receives a connection request from localhost with a message. If this message's topic is in the list of topics to be sent through the bridge, then this messages will be published in the Intermediate Broker.

D-THE WGD

The Wireless Gas Detector can now be powered on. Let's remember that the MQ2 Gas Sensor won't give accurate readings during initial warm-up process (a couple of minutes). Nevertheless, this shouldn't cause any problem, because also ESP8266-01 takes some time to be registered in the access point and to create a connection with the Local Broker. (Figure-10 illustrates this process).

Once the connections are made, the WGD starts subscribing topics in the Local Broker. In the Figure-34 this process is seen in the Arduino Ide Serial Monitor.

```

/dev/ttyUSB0

Connecting to TELUS1977
MAC address:
Connecting to AP
WiFi connected
P address:
92.168.1.11
MQTT connection
Attempting MQTT connection...failed, rc=-2 try again in 2 seconds
Attempting MQTT connection...failed, rc=-2 try again in 2 seconds
Attempting MQTT connection...connected
utTopic/23.17.224.11/192.168.1.11
Subscribe topics
ublish message: 1-Gas Sensor IP:192.168.1.11 MAC:AD:18:14:7F:CF:5C Reading value=1
utTopic/23.17.224.11/192.168.1.11
ublish message: 2-Gas Sensor IP:192.168.1.11 MAC:AD:18:14:7F:CF:5C Reading value=1
utTopic/23.17.224.11/192.168.1.11
ublish message: 3-Gas Sensor IP:192.168.1.11 MAC:AD:18:14:7F:CF:5C Reading value=1
utTopic/23.17.224.11/192.168.1.11
ublish message: 4-Gas Sensor IP:192.168.1.11 MAC:AD:18:14:7F:CF:5C Reading value=1
utTopic/23.17.224.11/192.168.1.11
ublish message: 5-Gas Sensor IP:192.168.1.11 MAC:AD:18:14:7F:CF:5C Reading value=1
utTopic/23.17.224.11/192.168.1.11
ublish message: 6-Gas Sensor IP:192.168.1.11 MAC:AD:18:14:7F:CF:5C Reading value=1
utTopic/23.17.224.11/192.168.1.11
ublish message: 7-Gas Sensor IP:192.168.1.11 MAC:AD:18:14:7F:CF:5C Reading value=1
utTopic/23.17.224.11/192.168.1.11
ublish message: 8-Gas Sensor IP:192.168.1.11 MAC:AD:18:14:7F:CF:5C Reading value=1
utTopic/23.17.224.11/192.168.1.11
ublish message: 9-Gas Sensor IP:192.168.1.11 MAC:AD:18:14:7F:CF:5C Reading value=1
utTopic/23.17.224.11/192.168.1.11
ublish message: 10-Gas Sensor IP:192.168.1.11 MAC:AD:18:14:7F:CF:5C Reading value=1

Exception (3):
pcl=0x4021a5ff epc2=0x00000000 epc3=0x00000000 excvaddr=0x40105882 danc=0x00000000

tx: sys
p: 3ffff540 end: 3ffffb0 offset: 01a0

Reset due exception
>>>stack>>>
ffff6e0: 116e6fe6 258b835c 37b884ee fccd146b
ffff6f0: ffa81cd6 99135307 47647153 f0bfc1b

```

Figure-35. Serial monitor of WGD in production.

As in Figure-10, Figure-35 includes the connection to the AP, but continues with the MQTT connection. Maybe because the Local Broker was busy or wasn't already up, the connection is retried. Once the connection is achieved (Figure-34 -New Connection), the topics begin to be published. It is important to highlight that the topics are published as long as the Local Broker is up. Although, it is not aware of any communication beyond that point, therefore it could be that the Remote Broker is not in service or the Bridge to the Intermediate Broker was not created either. In this scenario, the WGD continues with its work, despite the state of the rest of the system.

E-GASDETECTOR APP AND MYSQL

With the WGD subscribing topics in the Local Broker, and these topics bridged to the Remote Broker, the GasDetector.py program will subscribe to them and store the data in the database.

```

TOPIC:outTopic/23.17.224.11/192.168.1.11 payload:495-Gas Sens
or IP:192.168.1.11 MAC:AD:18:14:7F:CF:5C Reading value=0
495
Gas
192.168.1.11
AD:18:14:7F:CF:5C
0
2017-01-30 17:19:50
data was stored succesfully
TOPIC:outTopic/23.17.224.11/192.168.1.11 payload:496-Gas Sens
or IP:192.168.1.11 MAC:AD:18:14:7F:CF:5C Reading value=0
496
Gas
192.168.1.11
AD:18:14:7F:CF:5C
0
2017-01-30 17:19:51
data was stored succesfully
TOPIC:outTopic/23.17.224.11/192.168.1.11 payload:497-Gas Sens
or IP:192.168.1.11 MAC:AD:18:14:7F:CF:5C Reading value=0
497
Gas
192.168.1.11
AD:18:14:7F:CF:5C
0
2017-01-30 17:19:52
data was stored succesfully
TOPIC:outTopic/23.17.224.11/192.168.1.11 payload:498-Gas Sens
or IP:192.168.1.11 MAC:AD:18:14:7F:CF:5C Reading value=0
498
Gas
192.168.1.11
AD:18:14:7F:CF:5C
0

```

Figure-36. GasDetector.py screen output.

Figure-36 shows the GasDetector.py output in CentOS terminal, and Figure-37 show the same information already stored the MySQL database.

```

29093 | 499 | outTopic/23.17.224.11/192.168.1.11 | AD:18:14:7F:CF:5C | 1 | Gas | 2017-01-30 17:19:54 | Regular re
ding | 192.168.1.11 |
29092 | 498 | outTopic/23.17.224.11/192.168.1.11 | AD:18:14:7F:CF:5C | 0 | Gas | 2017-01-30 17:19:53 | Regular re
ding | 192.168.1.11 |
29091 | 497 | outTopic/23.17.224.11/192.168.1.11 | AD:18:14:7F:CF:5C | 0 | Gas | 2017-01-30 17:19:52 | Regular re
ding | 192.168.1.11 |
29090 | 496 | outTopic/23.17.224.11/192.168.1.11 | AD:18:14:7F:CF:5C | 0 | Gas | 2017-01-30 17:19:51 | Regular re

```

Figure-37. Select output.

As many messages were received by second from the same publisher (all of them with the same date-time), the Seq# is useful to identify the messages in the database records.

F-TALK TO THE WGD

IoT devices are capable of delivering information, but also they can receive instructions. As an example of this instructions, the Arduino code (thanks to the callback function), can subscribe to incoming topics. The security for this action is implemented using Username and Password authentication on the Local Broker, but it can also include TLS authentication and encryption. The program verifies the incoming message, and is expecting for:

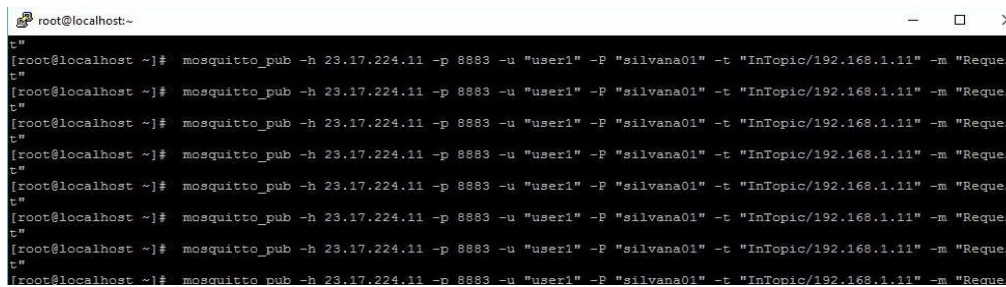
- “Request”, in response to this message the WGD will transmit the state of the MQ2 sensor in this right moment.

```
mosquitto_pub -h 23.17.224.11 -t "InTopic/23.17.224.11/192.168.1.11" -m "Request" -p 8883 -u user1 -P silvana01
```

- “Sleepn-“ where “n” is the gap in seconds between publishing two messages. Any value could be used. For example 1 = 1 message each second, and 0.01 is equal to 100 messages per second. In the following command the gap is fixed to 3 seconds.

```
mosquitto_pub -h 23.17.224.11 -t "InTopic/23.17.224.11/192.168.1.11" -m "Sleep3-" -p 8883 -u user1 -P silvana01
```

The Local Broker “shares” the message among all the IoT devices that are subscribing the topic (wildcards like # are used). The IoT device that subscribes the message, but in the topic finds its own IP, then takes the “Request” or “Sleep” order for itself.



```
root@localhost:~# mosquitto_pub -h 23.17.224.11 -p 8883 -u "user1" -P "silvana01" -t "InTopic/192.168.1.11" -m "Request"
root@localhost:~# mosquitto_pub -h 23.17.224.11 -p 8883 -u "user1" -P "silvana01" -t "InTopic/192.168.1.11" -m "Request"
root@localhost:~# mosquitto_pub -h 23.17.224.11 -p 8883 -u "user1" -P "silvana01" -t "InTopic/192.168.1.11" -m "Request"
root@localhost:~# mosquitto_pub -h 23.17.224.11 -p 8883 -u "user1" -P "silvana01" -t "InTopic/192.168.1.11" -m "Request"
root@localhost:~# mosquitto_pub -h 23.17.224.11 -p 8883 -u "user1" -P "silvana01" -t "InTopic/192.168.1.11" -m "Request"
root@localhost:~# mosquitto_pub -h 23.17.224.11 -p 8883 -u "user1" -P "silvana01" -t "InTopic/192.168.1.11" -m "Request"
root@localhost:~# mosquitto_pub -h 23.17.224.11 -p 8883 -u "user1" -P "silvana01" -t "InTopic/192.168.1.11" -m "Request"
root@localhost:~# mosquitto_pub -h 23.17.224.11 -p 8883 -u "user1" -P "silvana01" -t "InTopic/192.168.1.11" -m "Request"
root@localhost:~# mosquitto_pub -h 23.17.224.11 -p 8883 -u "user1" -P "silvana01" -t "InTopic/192.168.1.11" -m "Request"
```

Figure-38. Request made from a terminal connected to internet.

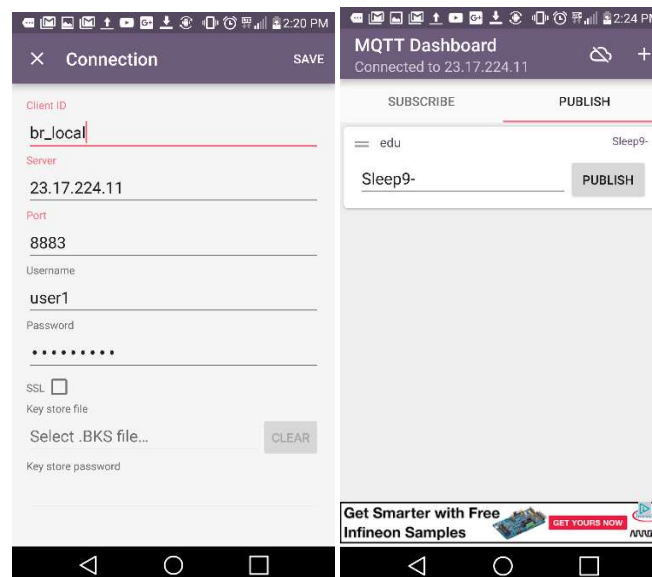
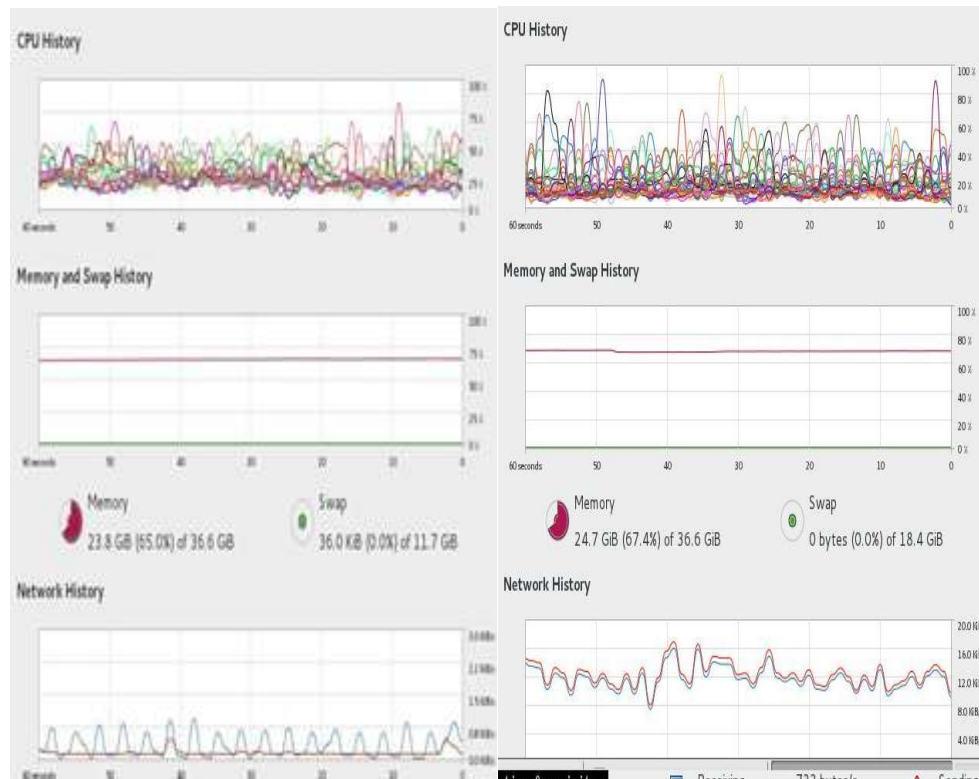


Figure-39. MQTT Dashboard. Publishes messages in Local Broker.

MQTT Dashboard (e.g.), an Android app, can publish/subscribe MQTT topics. It is SSL capable, although, we can not enable this feature on the Local Broker because this will cause that the IoT devices will be enforced to try SSL encryption.

PERFORMANCE

Using CentOS utilities and VMware performance monitor, some tests were done to different workloads. It is important to remember that we are working with only one IoT device. Despite we could simulate more traffic using applications, we would be using the same Local Broker and internet connection.



One message/second 1000 messages/second
Figure-40. CentOS/running OpenStack VM resources monitor.

Figure-40 shows a resource usage increase due Mosquitto increasing activity. Memory doesn't seem to be really affected by the higher traffic/operations by second, although, the Network usage itself and the CPU utilization show a substantial variation. So far, all the messages are arriving successfully and are stored by the database application without TLS.

A simple comparison made on VMware performance monitor, delivers similar results. On Figure-41, the CPU activity highly increased. Up to this moment no significant message loses were noticed and increasing the resources utilization should not be an alarm but a goal to achieve.

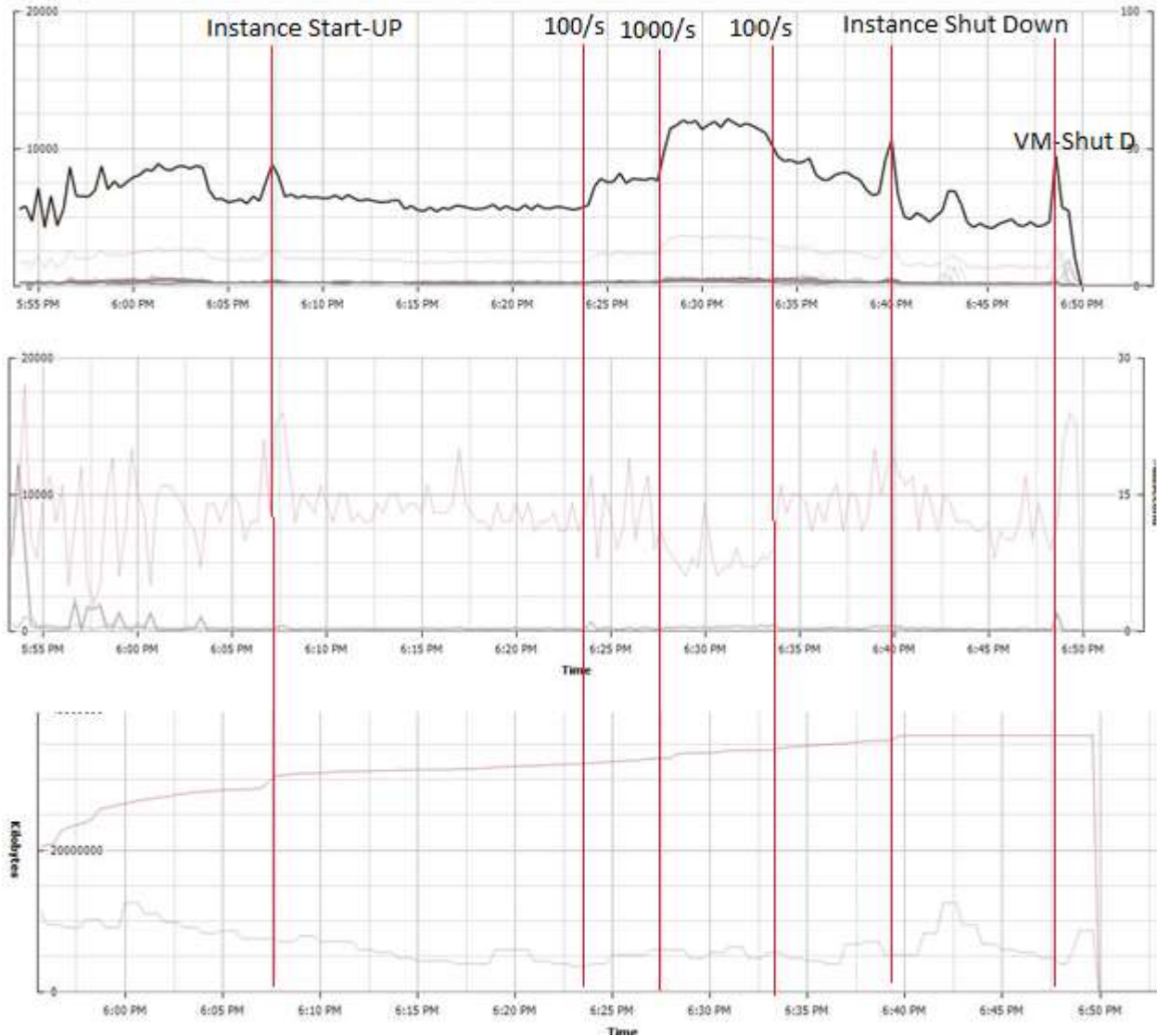


Figure-41. VMware resources monitor. Top= CPU, Middle= Storage, Bottom=Memory.

In Figure-41 there red lines show some changes due VM tasks. The highlighted events are:

- OpenStack Instance start-up
- Start receiving 100 messages/second.
- Receive 1000 messages/second.
- Receive 100 messages/second.
- Instance shutdown.
- OpenStack services stop and CentoOS shutdown.

Storage and Memory performance, upon the workload used, don't show any variation. Moreover, Disk usage was lower during the "high transmission/rate ". Memory shows an increasing Balloon Memory.²⁵ Therefore, this resources seem to be oversized for the required tasks and ready to

²⁵ "Virtual memory ballooning is a computer memory reclamation technique used by a hypervisor to allow the physical host system to retrieve unused memory from certain guest virtual machines (VMs) and share it with others."
<http://searchservervirtualization.techtarget.com/definition/memory-ballooning>

receive more workload. What requires attention is the CPU usage. The Figure-41 performance before implementing TLS. While 1000 unsecured messages/second were requiring 60% of CPU to be received processed and stored, after TLS was implemented, a higher usage of CPU was registered.

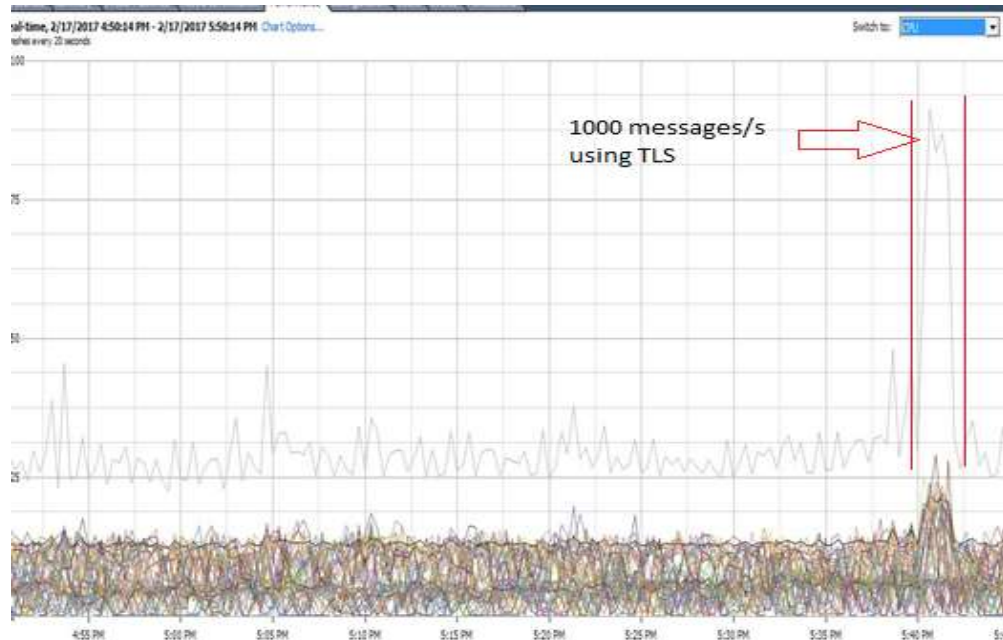


Figure-42. VMware CPU performance monitor. Using TLS

The same 1000 messages/second using TLS require 80% CPU. Figure-42 shows a peak caused for a change in the WGD that increased the rate to 1000 messages/second. During this test, all bridges were using TLS. The network usage for 1000 messages/sec is similar to the levels shown in the unencrypted tests (Figure-43).

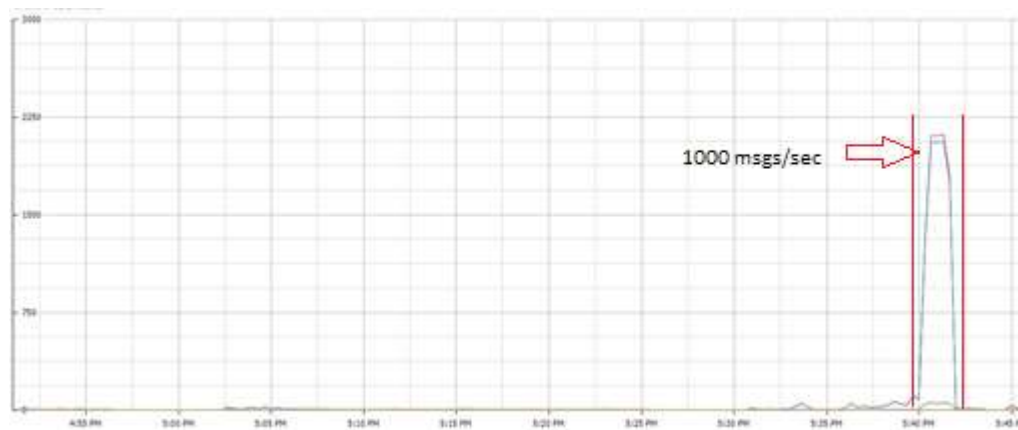


Figure-43. VMware Network performance monitor. Using TLS

FINAL CONSIDERATIONS

The objective of this project was to combine IoT with OpenStack cloud services, but the development process showed a wider researching field. Although the simplicity of IoT is the reason for its weakness, keeping it simple is the best way to make it more secure. But with simplicity we don't mean that IoT devices should make direct connections to servers. Some help is required.

The IoT device, by itself, can't provide much of security. Mosquitto is a powerful tool to communicate with IoT. Both, the device and the protocol can't deny their lack of complexity. Although, there is no reason to make things more complex.

Mosquitto already has embedded TLS features. The protocol provides enough tools to secure the communication process. As proposed in this project, the OpenStack instances can be used to provide computing and storage services to IoT, with totally isolated resources for different applications or customers, as well as being able to add more resources as the same pace the customers increase their requirements.

The knowledge acquired leads to this conclusion: the cleaner and simpler a system is, it will be easier to secure it. Stay simple is the best practice to close the door to intruders as well. But there is a challenge to this rule. 50 billion IoT²⁶ devices will be installed by 2020, and the temptation to reach them remotely is already enabled in people's mind. While the Remote Broker's side seems to be already secured, 50 billion backdoors could be exploited by botnets. With a little effort and investing some dollars per facility²⁷ we can install a similar structure for Local and Intermediate Brokers on site. The first one will deal with IoT devices and build a bridge to Intermediate/Remote brokers. The second one will authenticate (TLS) incoming topics from publishers in the internet to cloud services.

Given the possibilities that are arising with IoT, it is easy to wish everything will be as planned and start to seed IoT devices and directly connect them to start collecting data. But two Internet Security rules should be listened:

- 1) *"People make mistakes*
- 2) *The first rule will always be right."*²⁸

Imagine this: grandma's health monitor connected to her open (no password) Wi-Fi, plus a non-updated device's firmware or operating system, plus some low cost "smart" home appliances, plus a "home-made" IoT project to control the baby sitter, all of them using the same clouding service that a cloud provider offers to some applications of a research/energy plant. Now include the preceding two rules. Despite that IoT has a big potential, it is our responsibility to ensure that the technology is applied the right way and following the highest standards of quality.

In response to this concerns, when this project was starting, the first modification to the design was to add the Mosquitto Local Brokers. The WGD can communicate directly to the cloud, no doubt. But the Local Broker enables the coexistence of many IoT devices in the same facility and

²⁶ <https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/>

²⁷ Current price for a Raspberry Pi is US\$39.99, source www.ebay.com.

²⁸ Leonard Rogers, Internet Security, UofA-Mint. 2015.

addresses the main issue: the security of these devices. The Intermediate Broker comes to do the same on the Remote side.

Up to this date, there is not any published research/project proposing the brokers model that includes the usage of an Intermediate Broker (Figures 27-28). In fact, the idea of an Intermediate Broker is nothing more than a Local Broker that retransmits all the messages it is subscribing (from another broker). This straightforward solution becomes useful and has a lot of possibilities. The protocol itself figures out where the messages should be sent, the corresponding bridged Remote Broker, and this way the Remote Brokers are not exposed in a first line of fire against attacks, while everything is well secured using TLS encryption and authentication.

With Iptables port forwarding rules, we can't deploy as many servers (Remote Brokers) as desired. All the traffic goes in one direction. One alternative solution could be to subscribe to all topics in the first broker the messages find when they arrive, and an application could distribute messages using a database of topics and publishing in the Remote Brokers. It is using a Broker to receive the messages, decrypting and analyzing, to finally publish them in the right Remote Broker (quite the same!). But this could be rejected by customers that don't want their information to be decrypted and handled before it reaches their instance in the cloud. Customer's privacy, sensitive or financial information could be affected.

The Intermediate Broker does the same way as the Local Broker does. The Local Broker is our "trusted man in the middle" and provides protection to weak IoT devices. The Intermediate Broker is the guard of our OpenStack Instances that store all our information.

With the only requirement of using the port 8883 and TLS we are reducing backdoors and vulnerabilities coming from our system and we don't create any to other services.

Certainly, certificates and keys can be stolen as well as passwords might be brute-force decrypted. Attackers can pretend to be a genuine IoT device (belonging to our system), or one of the accepted Local Brokers. Although, IDS-IPS can be installed to "learn" normal traffic conditions and detect non-genuine messages.

Our Python application GasDetector.py currently discards any message without the right structure, but a more complex database design is recommended, where all the IoT devices should be registered in order to be able to validate all the publishers in our Brokers. Sequence beginning messages, control of sequences and logs auditory are required to make it even more secure. Also, because the simplest Local Broker has enough resources to, it is possible to develop local databases with a registry of authorized WGD publishing messages in each LAN.

TO BE CONTINUED

The following is a list of proposed tasks to continue with the current research:

- Study the WGD power requirements. Setup a battery powered circuit and establish how long can it operate without re-charging.
- Deploy Local and Intermediate Brokers in actual facilities. Enable TLS for internet publishers of topics oriented to modify the performance of the WGD.
- Deploy several devices in the same LAN and analyze the bandwidth requirements for different transmission rates. Create a function to calculate the required type of connection for certain number of sensors in one location.
- Deploy several Local Brokers using different internet connections.
- Deploy several Remote Brokers each one in a different OpenStack instances and segregate traffic using the topics configuration in the Intermediate Bridge setup.
- Create databases in the Local Brokers to store the information while it is impossible to connect to the Remote Brokers.
- Refine the ESP8266-01 Arduino code to verify that the warm-up time of the MQ2 sensor was reached. Because the time parameter is relative to environmental conditions, a temperature sensor could be used to verify that MQ2 is operating at $100\text{ C} \pm 20\%$.
- Combine sensors for different gases.
- Integrate LEDs and audible alarms to the circuit.
- Use the Analog Output of the gas sensors and report different concentrations of gas.
- Modify the circuit of the WGD to install backup sensors to avoid wrong values due malfunction.
- Modify the circuit to install two or more sensors (for different gases) in the same device.

APPENDIX A, BUDGET

DETAIL	Cost CAD\$
ESP8266 Serial WI-FI Wireless Transceiver Module Send	2.64
100Pcs NPN Transistor TO-92 2N2222A 2N2222	1.49
400X 0.25w 1/4w Metal Film Resistor Pack Kit 1%	4.67
210Pcs 25 Value 0.1uF~220uF Electrolytic Capacitors	4.69
MQ-2 MQ2 Gas Sensor Module Smoke Methane Butane Detection for Arduino	1.64
5V 3.3V FT232RL USB To Serial 232 Adapter Download Cable Module For Arduino	3.61
FT232RL 3.3V 5.5V FTDI USB to TTL Serial Adapter Module for Arduino Mini Port	2.48
Basic Starter Kit UNO R3 400 Breadboard LED Jumper Wire for Arduino TE132	22.32
5pcs 4.5V-7V to 3.3V AMS1117-3.3V Power Supply Module AMS1117-3.3	1.56
Soldering Kit	24.99
Female jumper cables	3.75
Solderless Breadboard Bread Board 830 Tie Points Contacts	3.19
LCD Digital Volt Ohm Meter Voltmeter Multimeter 830L	10.78
Total spent	87.81

APPENDIX B, TIMELINE

Start	Duration	Action/Task in progress
Sep 09	45 days	Order Parts (online).
Oct 06		First Pieces were received.
Oct 23		All pieces received. Build circuit.
Oct 23	5 days	ESP8266 is rebooting during Wi-Fi connection to AP because ESP8266 requires more energy in this stage. Solution: install 3 more capacitors, two after the AMS1117 regulator and a smaller one in the closest available connection to the ESP8266 power input.
Oct 29	7 days	Install local broker. Subscribing and publishing topics with test messages. Modify Arduino example code for ESP8266. Work on Mqtt connection, obtain ESP8266 MAC and IP, convert this information in different formats and use to build topics and messages.
Nov 06	4 days	Develop Python code to subscribe topics and store in MySQL in the Local Broker.
Nov 11	7 days	Install a test Remote Broker in own VM. Build bridge with local broker and run Python app to store data in Remote Broker.
Nov 19	10 days	Download and install different OpenStack deployments in own laptop. Tests with Autopilot and Devstack AIO and PackStack AIO setup. Expand laptop memory and install new hard-drive. Decide to use PackStack AIO on CentOS. Low number CPUs make impossible to create OpenStack Instances.
Nov 29	3 days	Test Android apps to publish topics in Local Broker, and test Callback performance.
Dec 3	1 month	Project on hold, waiting for Lab Server with public IP.
Jan 4	1 day	Access is granted to VPN and Server. Public IP is assigned.
Jan 6	7 days	Perform and verify different OpenStack configurations.
Jan 14	5 days	Create Remote Broker. Build bridge. Subscribe to topics published by WGD and store data in MySQL.
Jan 20	2 days	Enable Mosquitto username-password in local and remote brokers. Modify MQTT connection Arduino Code and mosquitto.conf files.
Jan 23	1 day	Begin final report preparation. Gather codes, screenshots. Unsecured communications tests.

Jan 25	13 days	TLS tests. Problems with SSL handshake. Connection timed-out. Problems with traffic arriving to public IP and forwarded to OpenStack Instance.
Feb 08	5 days	Implementation of Intermediate Broker. Tests with encrypted messages. Success!.
Feb 13	7 days	Report redaction.
Feb 22		Submit Report