



UNDERSTANDING Mr. MOSQUITTO



Rachen Ravichandran



UNDERSTANDING Mr. MOSQUITTO

An MQTT Primer



Rachen Ravíchandran

CONTENTS

1. Got a secret. Can you keep it?	1
What we are going to do in the rest of the tutorial.	
2. It all starts here	2
Understanding the basics of HTTP and MQTT.	
3. Setting up your server	11
Installing mosquitto and using it for the first time.	
4. Creating our Hangman Chat Room	19
Installing Paho MQTT Python Client and understanding its functions.	

CHAPTER 1

GOT A SECRET, CAN YOU KEEP IT?

MQQM is a top notch web development company. It is *THE* company where every person in this world dreams to work in. There is Mr Mosquitto's couch in which you can crash in for late night stay or Mrs Eclipse's free pizza shop where you can devour anything you want or Mr Hivem's video gaming console with which you can play when you are bored.

Free game and free food? Who wouldn't want that??? And you... you want it so badly, that you had written a fake resume stating that you know all about IoT and are the top web developer of the century and *even managed to get in*.



Tarry a little! Don't be so happy. Your first project is up. You need to develop a chat room software where every employee can chat with everyone else in the company. They even want you to add a game to this chat room. They want the lights to turn off when someone says OFF in the chat, doors to close when someone says CLOSE and all those IoT stuff. And your deadline is nearing. Your brain boils; heart races; and legs shiver. What are you going to do? (Moral: No fake resume.)

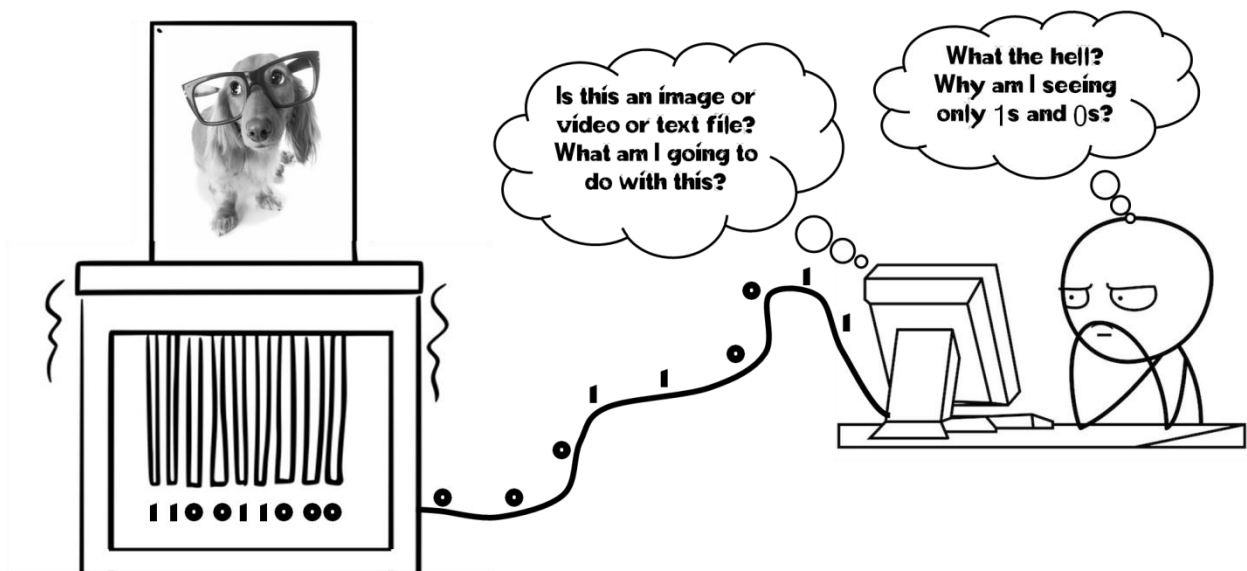
Worry you not! We got a handy tool to do all this and I'll tell you how. It'll be our little secret.

CHAPTER 2

IT ALL STARTS HERE

You cannot send your IoT or chat room data over the internet just like that and using internet means using some sort of technology to transmit your information from one computer or machine to some other computer or machine in the world (we'll restrict ourselves to the company for now.) Then, how to transmit your data?

For example, you might want to send a cool image of your dog wearing nerdy glasses to your colleague, Mr Pippy, over the internet. Let's assume you do this by a simple cable. Your image is converted to 1s and 0s and transmitted via this simple electric cable to Mr Pippy's computer. But how will your colleague's computer know that the 1's and 0's you are sending is neither a text file nor a video file but is an image file? If this cable is connected to two or more of your colleague's computers, which computer should the image go to?



This is where internet protocols come into the picture. They make sure your data reaches Mr Pippy. There are many internet protocols but the most commonly used is the notorious HTTP – Hyper Text Transfer Protocol.

Note: There are many layers in the internet. HTTP belongs to the Application layer of the OSI model of networking

How does HTTP protocol send your data? Each device on the internet is identified by an address called IP address. So, the computer to which you are going to send your data to is identified by its host address (looks like 168.10.2.1) or domain name if one exists (e.g., google.com.) Thus, HTTP protocol in your computer adds this host address, the type of data you are sending (image or HTML or JS etc.), the date and time and all other related stuff to your original data (this additional information is called a header as in Fig. 2.1.) It is then taken to the recipient device or server which reads the header to understand the type of information you've sent.

```
Access-Control-Allow-Origin: *  
Cache-Control: max-age=604800  
Connection: keep-alive  
Content-Encoding: gzip  
Content-Type: image/svg+xml  
Date: Wed, 03 Feb 2016 17:29:50 GMT  
ETag: W/"552f8e59-639"  
Expires: Wed, 10 Feb 2016 17:29:50 GMT  
Last-Modified: Thu, 16 Apr 2015 10:26:33 GMT  
Link: <https://www.keycdn.com/img/cdn-fast.svg>; rel="canonical"  
Server: keycdn-engine  
Strict-Transport-Security: max-age=31536000; includeSubdomains; preload  
Transfer-Encoding: chunked  
Vary: Accept-Encoding
```

Fig. 2.1 A sample HTTP header

This process has been over-simplified; but, there are other underlying technologies in the internet.

Consider another example. This time you'd like to google dog images. When you type 'www.google.com' in your browser, you send a request for HTML page of the Google from your computer to Google's computer (or server.) To do this you establish a connection with the host and send your request with the appropriate header to the server. Google server responds to this request and sends its data along with related header to you and the connection is disconnected. But only a part of the webpage is sent before the connection is lost and so every time you request for the other parts of the web page, your computer keeps reconnecting, requesting, receiving and disconnecting with the server in the same order.

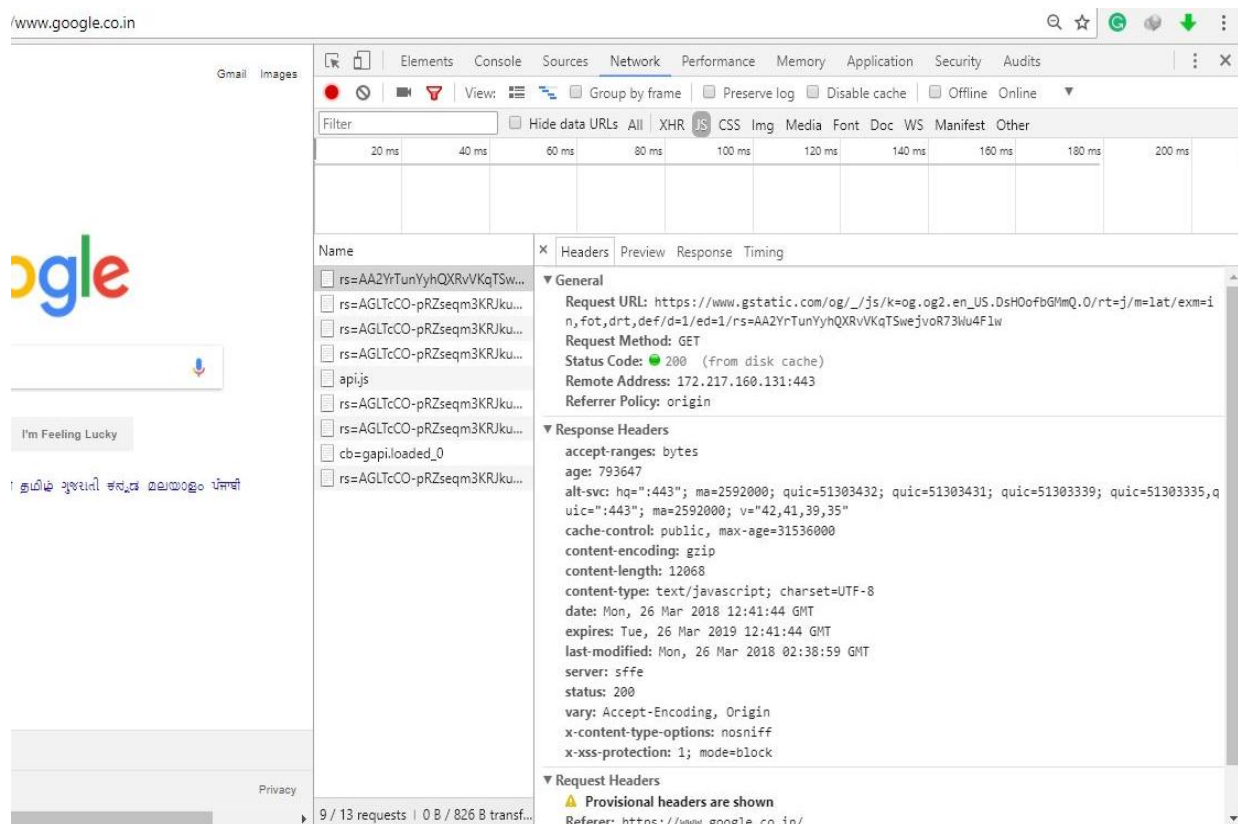


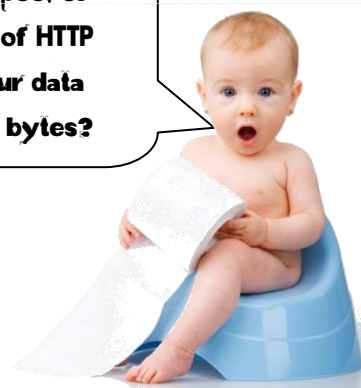
Fig. 2.2 Pressing F12 in Chrome after loading www.google.com shows the above request/response header

To understand more, head over to your browser and type www.google.com. Press F12 (in chrome browser) and click 'Network' tab. You'll find a list of request and response headers (all for a single goddamn web page!!!) and each one requires a separate connection and disconnection to the server (Fig. 2.2.)

Even if your data is hardly few bytes, your headers themselves might carry information for several bytes; for e.g., in a pool of 666 bytes your data might be 6 bytes. If you are going to use devices like your mobiles, microcontrollers or embedded systems, where memory is severely constrained, battery consumption must be as low as possible and network usage should be minimum (i.e., your internet connection speed is low), using HTTP protocol becomes a serious disadvantage.

Incidentally, most of your IoT projects use such constrained devices. For e.g. a temperature sensor which sends the temperature value to the internet via a WiFi module cannot afford a power-hungry protocol draining its battery or stripping away its memory as implementing HTTP might affect the performance of the device. Moreover, your HTTP requests and responses are done from one device to another device only, i.e. on a one to one basis. But, what if you want your temperature sensor to send the data to your mobile phone, your computer and your e-mail id? It has to request a connection with the 3 devices one at a time, append the temperature value with several bytes of header, send it to each device and disconnect them.

OMG! In a pool of 666 bytes of HTTP header your data might be 6 bytes?



Now, do you really think that HTTP protocol is ideal for your IoT projects? Or you would like to kick it and start using something more convenient and ideal? But, what would you use?

MQTT... MQ Telemetry Transport protocol (or Message Queuing Telemetry Transport - there's really no proof that this is the right expansion for MQTT though some people might expand it that way. But, MQTT has nothing to do with message queuing)

Terribly Boring Textbook Definition

MQTT is a machine-to-machine (M2M)/Internet of Things connectivity protocol. It was designed as an extremely lightweight publish/subscribe messaging transport. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is premium. *-Wikipedia*



For now, ignore this boring text. We'll get into important features of MQTT before analyzing the definition. M2M is a subset of IoT where small number of devices communicate with each other, while IoT involves communication of a device with a large number of other devices, sometimes at cloud level (some might defer with my explanation but at the end of the day, all of them are methods of connecting one device to several others.) So, MQTT protocol is basically a protocol for IoT or M2M.

MQTT protocol is built on the grounds of PUBLISH/SUBSCRIBE concept. This is basically like your Facebook (Fig. 2.3.) You *subscribe* to your friends. When your friends post something, you see the posts in your timeline. If your friends had subscribed to you, your posts are *published* to their timeline and it is just not one friend, but every friend receives your post. The Facebook server manages to whom the posts should be sent i.e., from one friend (a publisher) to all his/her friends (subscribers.)

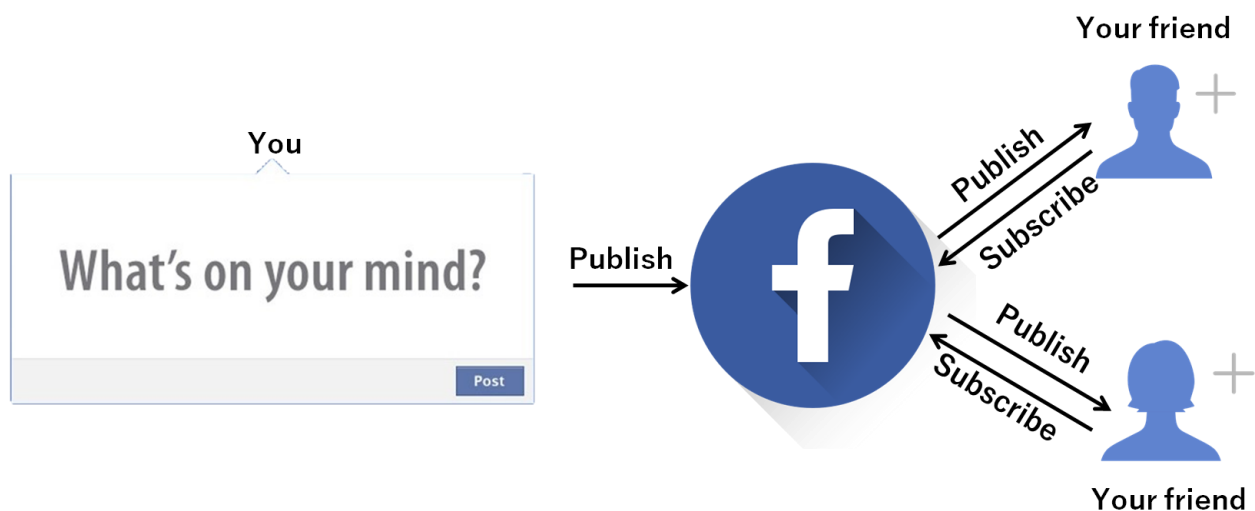


Fig. 2.3 Publishing your posts in Facebook timeline

This is how MQTT protocol works, except friends in MQTT are topics. Topics are strings separated by '/' character to identify the type of message. For example, if temperature sensor publishes its data to the topic '*sensor/temperature*', all devices subscribed to this topic receives the corresponding data. But who receives the data published and sends all these data to the subscribers? MQTT broker/server takes care of that. They are basically web services who receive the publisher's messages and pass them to

the subscribers. There are many web services which provide MQTT servers at premium cost or for free e.g., Mosquito broker, Eclipse IoT broker, HiveMQ broker, AWS IoT etc.

So, when a device publishes or subscribes to a topic, it maintains a connection with the broker until a voluntary disconnect command is given or network failure occurs (at broker side or the device side.)

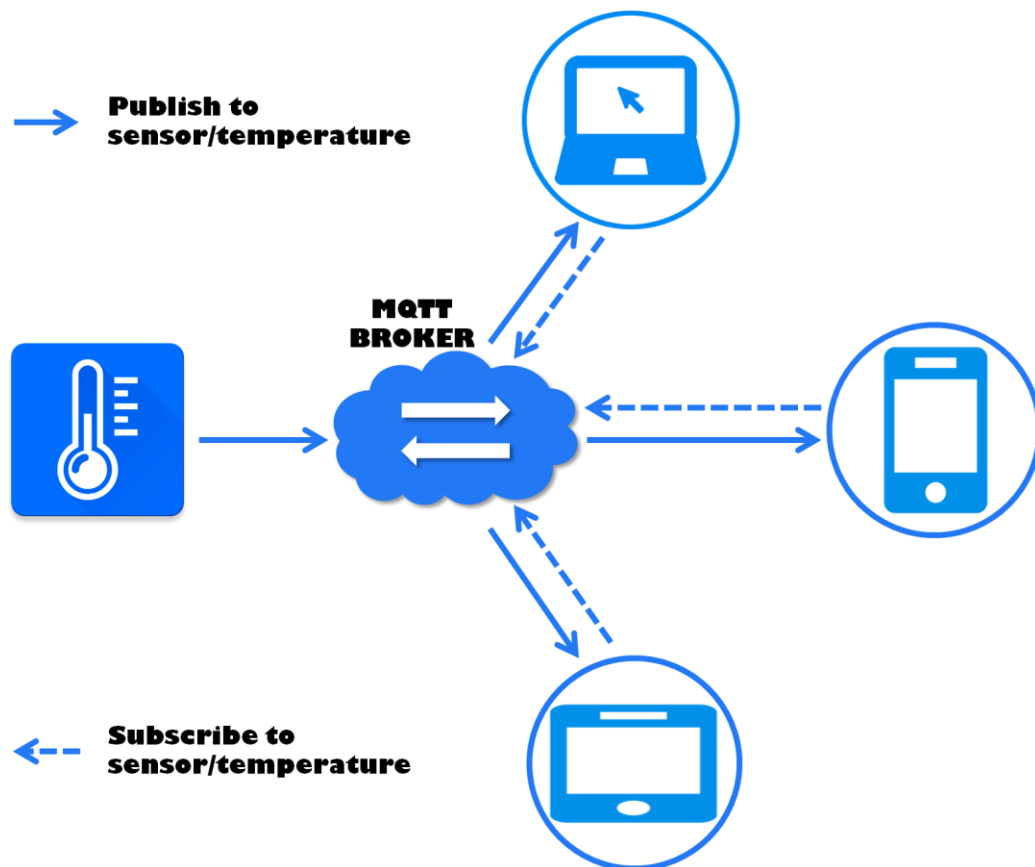


Fig. 2.4 Publishing your temperature to topic sensor/temperature

When the temperature sensor wants to publish a message to the topic 'sensor/temperature' (Fig. 2.4), it connects with the broker first and then publishes the message to it. Unlike HTTP, it does not disconnect with the broker. Once, the message is received the broker may or may not send an

acknowledgement message (indicating that the message is received) to the publisher. The broker then sends this message to all the devices subscribed to the topic. Similar, to the publisher, all the subscribers maintain their connection with the broker. So, a multiple response/request connection is not made as in HTTP.

Both the publishers and subscribers are not directly connected to each other by any means but only to the broker. Moreover, the subscriber can subscribe to a topic even before it is published. If you look at the subscribe or publish message, the additional information (i.e. header) is of 2 bytes contrary to several bytes of header in HTTP. Unlike HTTP, making individual connections with each device several times, MQTT makes a single connection with the broker and the rest is managed by the broker which can be a remote server. Thus, it absolutely suits IoT applications due to their low network and memory usage and as a consequence of which lower battery consumption.

Note: This doesn't mean you should abandon HTTP altogether. It is one of the best and secure protocols out there, and MQTT will never replace it. But it is just not right for all your IoT projects.

Going back to the definition, MQTT protocol can be summarized as:

1. *machine-to-machine (M2M)/"Internet of Things" connectivity protocol* – suitable for M2M or IoT applications.
2. *extremely lightweight* – one-time connection to broker and lower battery usage
3. *publish/subscribe messaging transport* – one-to-many model

4. *small code footprint* – header is of 2 bytes and code requirements are lesser
5. *premium network bandwidth* – one-time connection to broker and lower header size; thus, extremely reduced network usage and quicker message transmission even in slow network connectivity speed

In the next chapter, we'll start implementing MQTT and get into our first project.

CHAPTER 3

SETTING UP YOUR SERVER

There are many MQTT applications out there. Here, we'll setup our own broker called 'mosquitto' on our computer and use one of several MQTT apps from Google Play Store. Let us install 'mosquitto'. There are two ways to do it.

METHOD 1: Installing mosquitto for Windows - the hard way

Installing mosquitto requires installing dependencies as in Fig. 3.1 (additional files are to be installed). Skip to method 2, if you don't want the hassle of installing it the hard way (especially for beginners.)

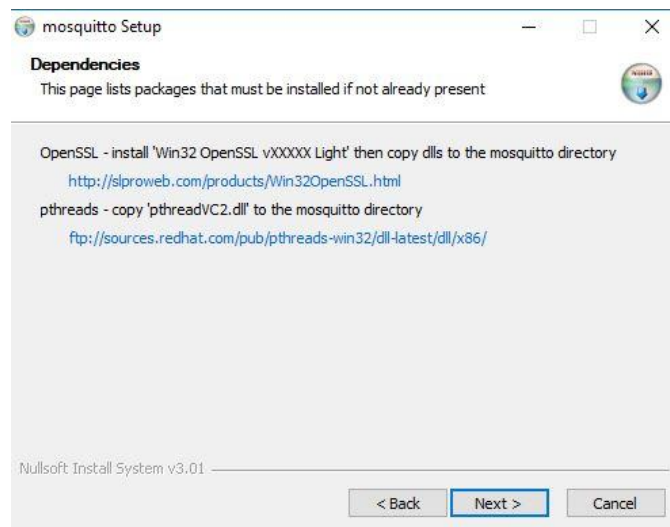


Fig. 3.1 Dependencies for your mosquitto broker

Step 1: Go to the [link](#), select '*mosquitto-x.x-install-win32.exe*' under Windows section and download the file.

Step 2: Now install your mosquitto broker in any destination folder you want. Click 'next' through various step whenever required.

Step 3: Go to the [link](#), select '*Win32 OpenSSL v1.0.2o Light 2MB Installer*' and install it in a suitable destination folder (note the destination folder.)

Step 4: Go to the destination folder where you have installed OpenSSL (mostly *C:\OpenSSL-Win32*), copy the files '*libeay32.dll*' and '*ssleay32.dll*' and paste them in to the mosquitto folder (generally, mosquitto is installed in *C:\Program Files (x86)\mosquitto*)

Step 5: Now, go to this [link](#), download the file named '*pthreadVC2.dll*' and copy it to the mosquitto installation folder.

Step 6: Reinstall mosquitto to same destination folder using the exe file from Step 1.

METHOD 2: Installing mosquitto for Windows - the easy way

Copy the '*mosquitto*' folder provided into the suitable destination folder. That's it! Quick and easy. **A word of caution:** the folder contains mosquitto broker of version 1.4.15 and if you wish to upgrade it, you can find it on their [web page](#). For installation of mosquitto in other platforms, visit this [link](#).

For using MQTT in your mobile phone, there are many apps out there. I would recommend you to install '*MyMQTT*' app from Google Play Store. You can use other apps too whichever suits your requirement.

Mosquitto is an MQTT broker as well as a client. You can publish or subscribe your messages to a topic in command line using mosquitto. To start mosquitto in your computer, search for '*services*' in your start menu and find '*Mosquitto Broker*' (Fig. 3.2 as in next page.) Most probably your mosquitto server might be running already. Select '*Start*' by right-clicking it, if it doesn't.

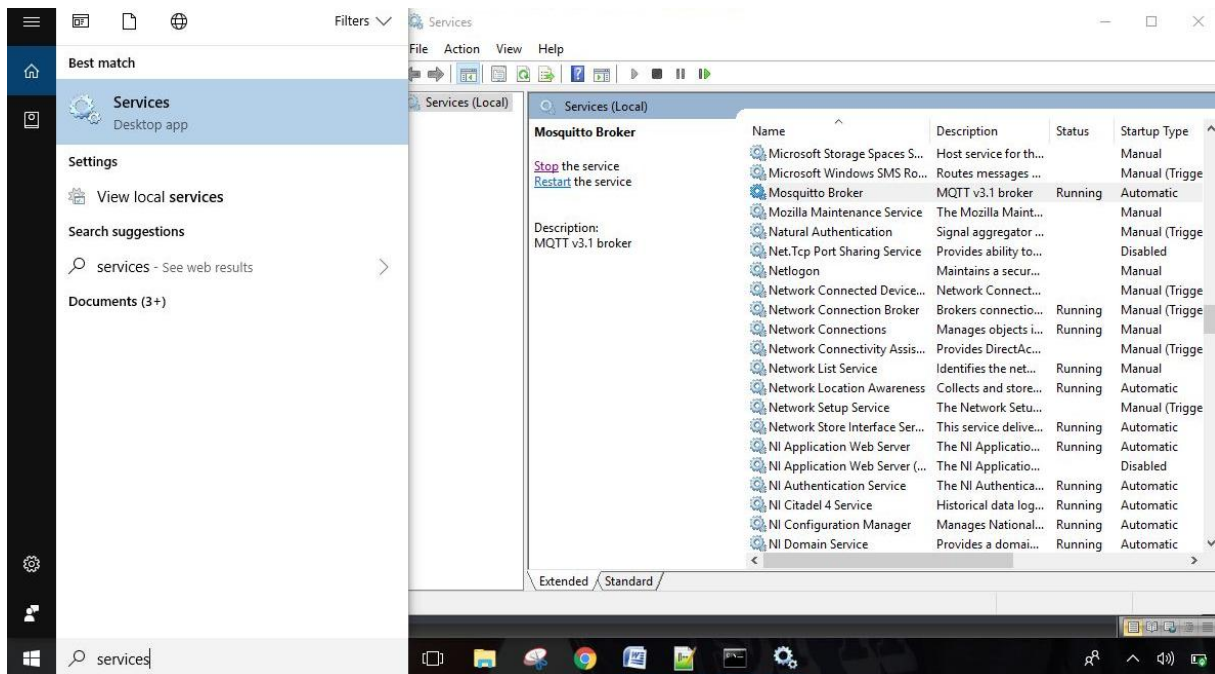


Fig. 3.2 Starting your mosquitto service

Open your command prompt and change your directory to the folder where you've installed mosquitto files using `cd` command as given below:

```
C:\Users\rache> cd c:\Program Files (x86)\mosquitto
c:\Program Files (x86)\mosquitto>
```

Note: To use mosquitto, always make sure that you are in the mosquitto directory.

Let's subscribe to the topic 'hello' so that we can hear any messages from the publisher. Type the following in your command prompt:

```
mosquitto_sub -t hello
```

'mosquitto_sub' is the command used to subscribe to the topic 'hello' indicated after the `-t` option. Now open a new command prompt and in your mosquitto directory, type the following:

```
mosquitto_pub -t hello -m "hello world"
```


'mosquitto_pub' publishes the message "hello world" specified after the -m option to the topic 'hello' and you'll see this message in your subscribed command prompt (Fig. 3.3.)

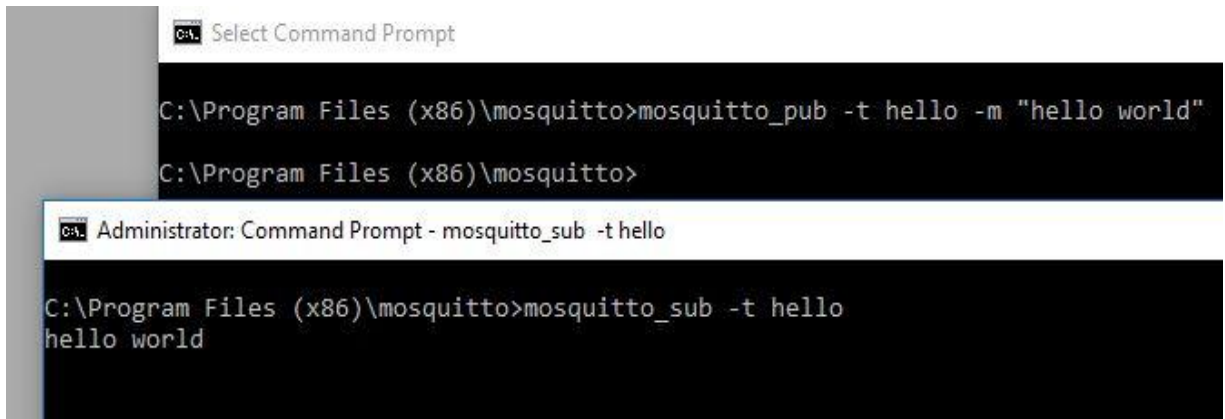


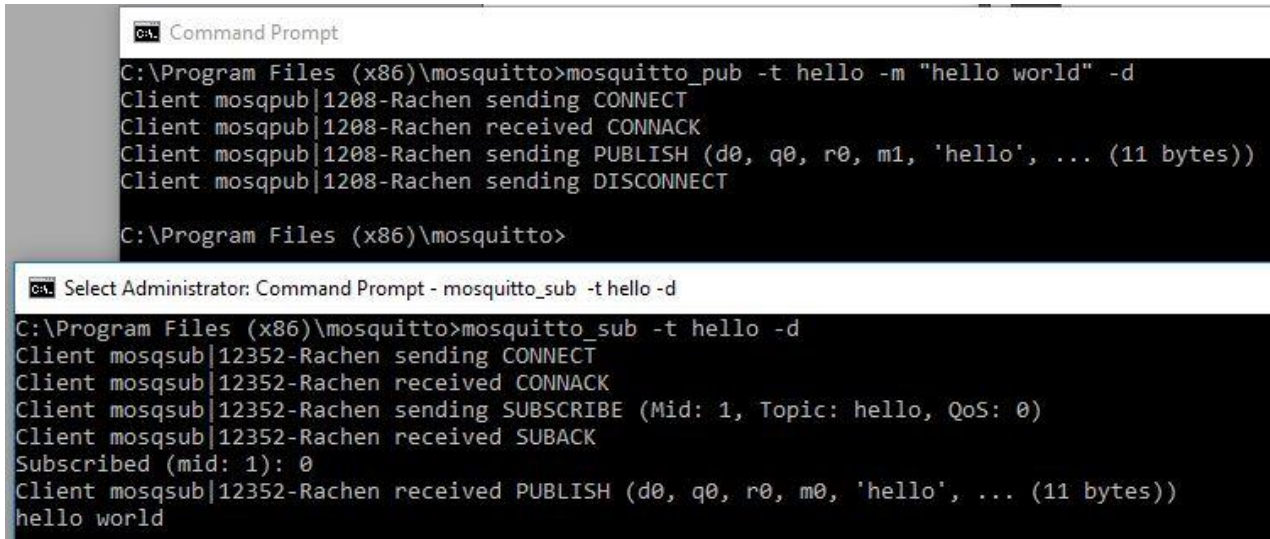
Fig. 3.3 Publishing and subscribing to a topic

The mosquitto broker receives this message from the publisher and sends it to the subscribers. Note that the message is not directly published from the publisher to the subscriber even though it might look like that. Your mosquitto broker running in the background handles the message. Try other messages too and observe the output received in mosquitto_sub command prompt. Also, try opening multiple mosquitto_sub command prompts.

mosquitto_sub keeps running in your command prompt to receive any new messages. Press 'Ctrl + C' on your keyboard to interrupt your subscription. Add -d option to mosquitto_sub and mosquitto_pub commands as below:

```
mosquitto_sub -t hello -d  
mosquitto_pub -t hello -m "hello world" -d
```

This will let us print the debug messages so that we can understand what really is happening behind the process (Fig. 3.4.)



The image contains two screenshots of a Windows Command Prompt. The top screenshot shows the execution of the `mosquitto_pub` command with the `-d` flag, resulting in debug output for a client connecting, sending a PUBLISH message, and disconnecting. The bottom screenshot shows the execution of the `mosquitto_sub` command with the `-d` flag, resulting in debug output for a client connecting, subscribing, receiving a PUBLISH message, and printing the message content.

```
C:\Program Files (x86)\mosquitto>mosquitto_pub -t hello -m "hello world" -d
Client mosqpub|1208-Rachen sending CONNECT
Client mosqpub|1208-Rachen received CONNACK
Client mosqpub|1208-Rachen sending PUBLISH (d0, q0, r0, m1, 'hello', ... (11 bytes))
Client mosqpub|1208-Rachen sending DISCONNECT

C:\Program Files (x86)\mosquitto>

C:\Program Files (x86)\mosquitto>mosquitto_sub -t hello -d
Client mosqsub|12352-Rachen sending CONNECT
Client mosqsub|12352-Rachen received CONNACK
Client mosqsub|12352-Rachen sending SUBSCRIBE (Mid: 1, Topic: hello, QoS: 0)
Client mosqsub|12352-Rachen received SUBACK
Subscribed (mid: 1): 0
Client mosqsub|12352-Rachen received PUBLISH (d0, q0, r0, m0, 'hello', ... (11 bytes))
hello world
```

Fig. 3.4 Using `-d` option in mosquitto

Take a look at the `mosquitto_pub` command prompt. You can see that the client first establishes a connection to the mosquitto broker by sending a `CONNECT` message. Here, the broker resides in your local computer itself at `127.0.0.1` which is also referred to as the local host. The broker then sends an acknowledgement to your `CONNECT` message, as a `CONNACK` message. Once the connection is established, the `mosquitto_pub` client sends your message as `PUBLISH` message. Finally, a `DISCONNECT` message is passed to indicate the end of the message transfer. Similarly, you'll find `CONNECT` and `CONNACK` messages in your `mosquitto_sub` window. `SUBSCRIBE` and `SUBACK` represents subscribe and subscription acknowledgement messages respectively. 'Mid' represents message id which is the number of messages received. The next line (`PUBLISH`) indicates the header received by the `mosquitto_sub` service.

Another important parameter in MQTT is 'QoS' or 'Quality of Service'. It indicates the certainty that your message is transmitted to the broker and received by the subscriber. There are 3 levels of QoS as below:

1. **QoS 0:** In this, the message is transmitted once and doesn't care if it is received by the broker or subscriber client. For example, the topic *'hello'* publishes the message "he11o". If this message does not reach the broker due to network error, the publisher will not know that the message is transmitted properly and the subscriber will not get the message. If a message is lost, it is lost forever and the publisher will begin to send the next message. This QoS is selected if your application is not so strict about receiving all the messages. Most of the MQTT clients (e.g. mosquitto) defaults to QoS level 0.
2. **QoS 1:** In this, the same message is transmitted at least once (i.e. one or more times). The publisher or broker keeps transmitting the message until an acknowledgement message is received. As a result, there is a possibility of receiving duplicate messages but there is also the certainty of receiving the message properly.
3. **QoS 2:** In this, a message is transmitted only once by a series of acknowledgement signals. So, this level is relatively slow but extremely accurate in getting your message across.

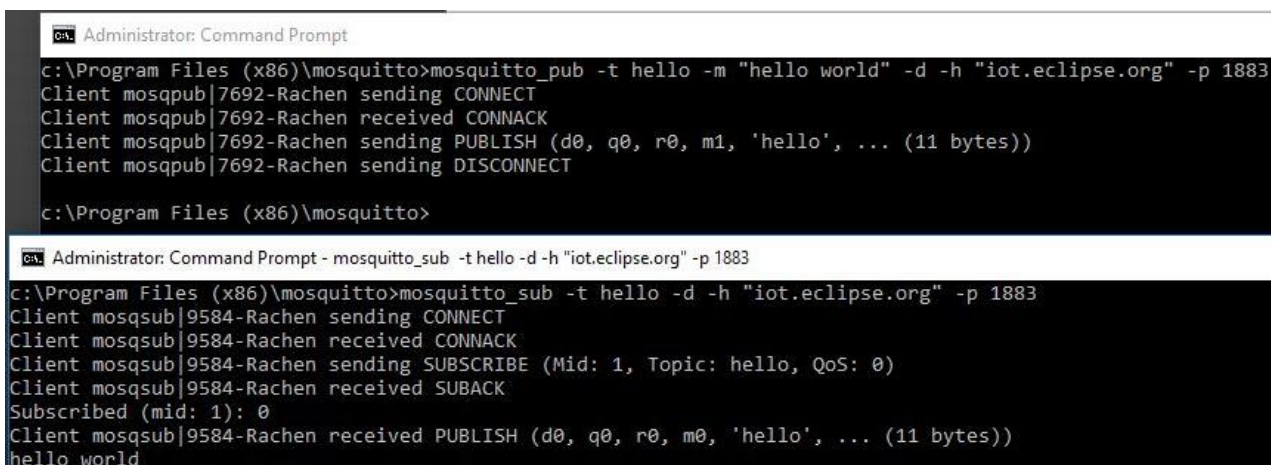
To check how this works, add `-q` option followed by the level to `mosquitto_sub` and `mosquitto_pub` commands as below:

```
mosquitto_sub -t hello -d -q 2  
mosquitto_pub -t hello -m "hello world" -d -q 1
```

Try for different values of QoS using `-q` option and observe the output.

All the above methods are suitable only for receiving messages in your local computer. What if we want to receive the message in some other computer or mobile phone? This is where other MQTT brokers come in. You can use any MQTT broker as you wish. We'll use IoT Eclipse broker at the host address `'iot.eclipse.org'` as the MQTT broker (Fig. 3.5.) Mosquitto defaults to local host broker. To specify the broker explicitly, add your host address after `-h` option as below:

```
mosquitto_sub -t hello -d -h "iot.eclipse.org" -p 1883
mosquitto_pub -t hello -m "hello world" -d -h
"iot.eclipse.org" -p 1883
```



The image contains two screenshots of a Windows Command Prompt window. The top screenshot shows the execution of the `mosquitto_pub` command with the following output: `Client mosqpub|7692-Rachen sending CONNECT`, `Client mosqpub|7692-Rachen received CONNACK`, `Client mosqpub|7692-Rachen sending PUBLISH (d0, q0, r0, m1, 'hello', ... (11 bytes))`, and `Client mosqpub|7692-Rachen sending DISCONNECT`. The bottom screenshot shows the execution of the `mosquitto_sub` command with the following output: `Client mosqsub|9584-Rachen sending CONNECT`, `Client mosqsub|9584-Rachen received CONNACK`, `Client mosqsub|9584-Rachen sending SUBSCRIBE (Mid: 1, Topic: hello, QoS: 0)`, `Client mosqsub|9584-Rachen received SUBACK`, `Subscribed (mid: 1): 0`, and `Client mosqsub|9584-Rachen received PUBLISH (d0, q0, r0, m0, 'hello', ... (11 bytes))` followed by the message `hello world`.

Fig. 3.5 Publishing and subscribing using IoT Eclipse Broker

In practice, there are many protocols. How does your recipient computer know the protocol adopted by the sender? Port is used for this purpose. It is not a physical wire but actually a software method of telling which process or protocol is to be used. The default port for HTTP is 80. Since we are using MQTT protocol we'll use its corresponding port 1883 indicated by the `-p` option. Usually `-p` option is not required as it defaults to 1883.

Now, you can access your message from anywhere in the world. To check this, open your 'MyMQTT' app from your mobile phone and enter the details as in Fig. 3.6.



Fig. 3.6. Subscribing to a topic using MyMQTT app

Publish your message in mosquitto using IoT Eclipse broker as in Fig. 3.5. Your app's dashboard and mosquitto_sub command prompt will receive this message. Now, click publish tab in your app and enter the topic name and message you want to publish and observe the output in your app as well as the command prompt. Here, the IoT Eclipse broker handles your message from mosquitto in your computer and the app in your mobile phone. Now that we have a basic idea of MQTT, let us start making a chat application.

CHAPTER 4

CREATING OUR HANGMAN CHAT ROOM

Our first project is to create a chat application where any of the messages you send are published in the topic *'chatRoom/<your-name>'* using IoT Eclipse broker. The application should also subscribe to the topic *'chatRoom'* so as to receive any messages published by the master of the chat room. This is extremely simple with mosquitto broker, but what we expect is a GUI window instead of a gloomy black command prompt window. Moreover, instead of typing lengthy mosquitto commands every time we are using the chat application, we might need an alternative method. Paho MQTT Client tool is right here to save your job (literally.) Paho MQTT Client is a python library to simplify our MQTT process.

Installing Paho MQTT Client tool for Python:

Step 1: To use Paho MQTT, you need to first install Python on your system. Go to this [link](#), download the python file according to your computer specifications and install it. Skip if you have already installed python in your computer.

Step 2: Now copy the *'get-pip.py'* file into any folder you wish to. Open the command prompt and go to the directory where you've copied the python file using cd command. Then, type the following command:

```
python get-pip.py
```

Pip is a python package installer tool, similar to your windows installer, which installs the python packages into your system without any hassle.

Step 3: Type the following command in the command prompt:

```
pip install paho-mqtt
```

We are now ready to use Paho MQTT Client for our application. Let us start our first chat application given in '*client.py*' file. Right-click the file and select "Edit with Python IDLE". You'll find the code for the chat application. All necessary explanations for the code are added in the comments. Since we are using Paho MQTT library we need to import it using the following command:

```
import paho.mqtt.client
```

To create a GUI environment, we can use the python library 'Tkinter' which provides various GUI elements like windows, buttons, text entry fields, display labels etc.(called widgets as in Fig. 4.1.)

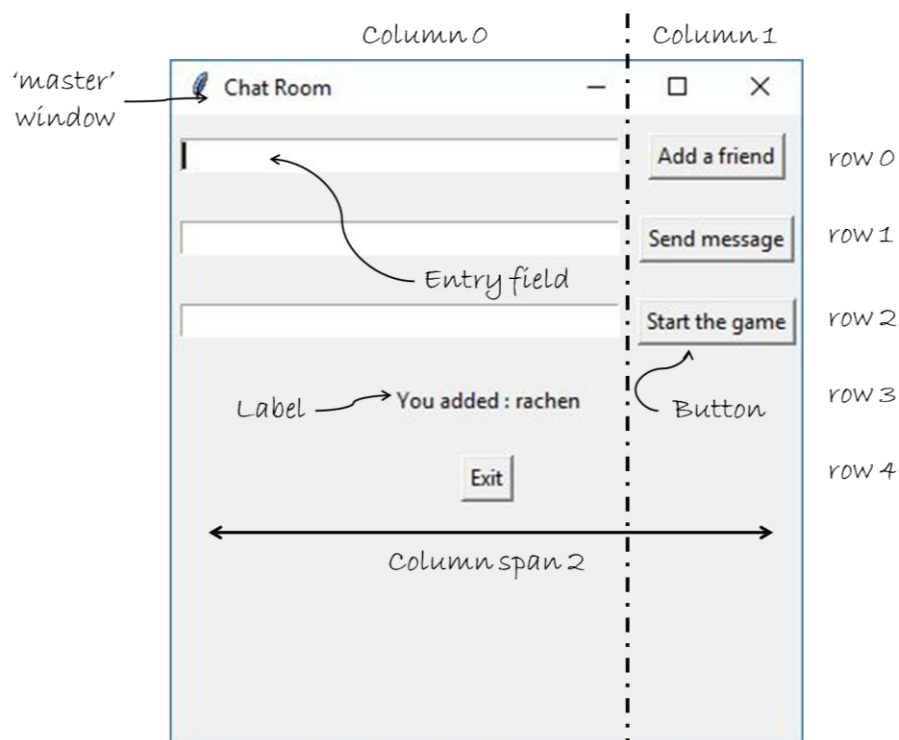


Fig. 4.1 Understanding Tkinter GUI

The window in which all the widgets are placed is divided into rows and columns called '*grid*'. The `grid()` function represents the row and column

number to which the widget is to be added, and other details. For more information, refer the comment lines or [Tkinter documentation](#).

To use MQTT client in your code, we need to define an object variable of type `paho.Client` which is done as follows:

```
client = paho.Client()
```

The following are some of the Paho MQTT functions (but not the complete list) which are employed in our code and are the most commonly used ones in MQTT:

```
1. client.connect("iot.eclipse.org", 1883)
```

This function will establish a connection with the host *"iot.eclipse.org"* at the port 1883.

```
2. client.subscribe(topic = "chatRoom", qos = 2)
```

This function will subscribe to the topic *'chatRoom'* with QoS = 2.

```
3. client.publish(topic = "chatRoom", payload = "hello", qos = 2)
```

This function will publish the message *"hello"* to the topic *'chatRoom'* with QoS value of 2.

```
4. client.loop_start()
```

This function establishes a network connection with the broker and the connection is maintained thereafter.

```
5. client.loop_stop()
```

The MQTT client is disconnected from the MQTT broker by sending a DISCONNECT message to the broker using this function.

Note that both publisher and subscriber are referenced to as MQTT client. For further information on Paho MQTT client library, visit this [link](#).

Run the *'client.py'* file by double-clicking it. It opens a GUI environment as in Fig. 4.2. Enter your name first and you can see that the button and entry field disables or freezes after entering your name indicating that you are ready to publish to the topic *'chatRoom/<your-name>'*.

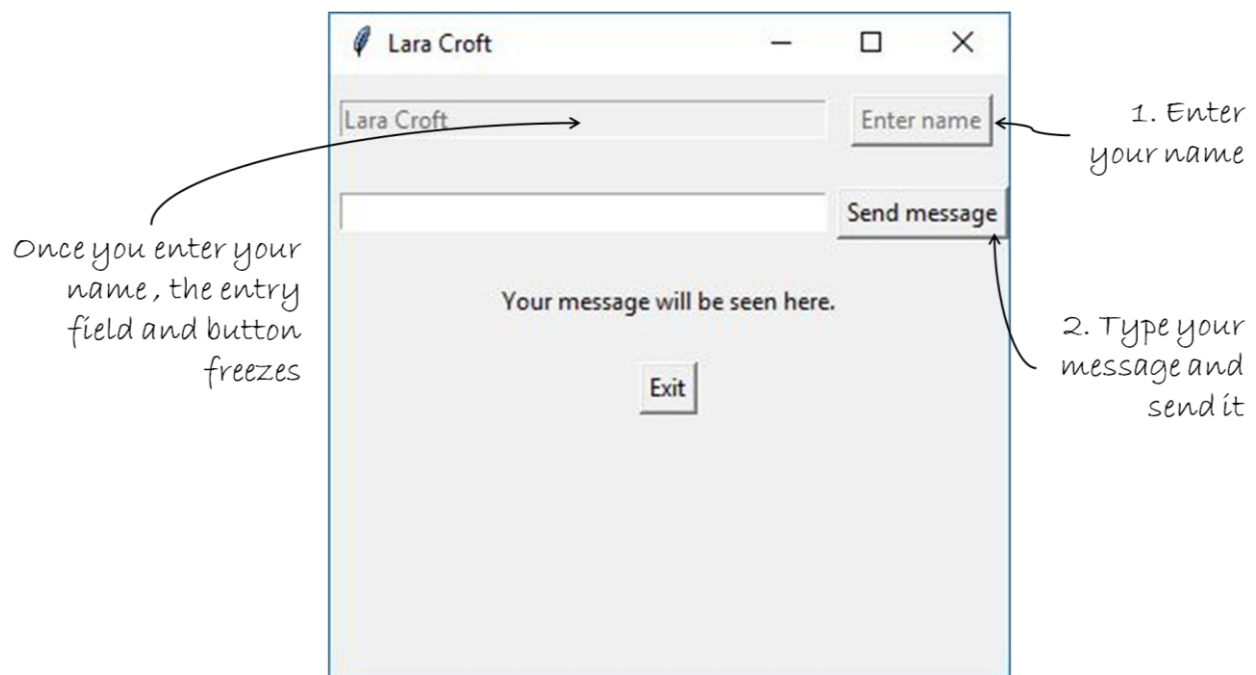


Fig. 4.2 *client.py* GUI window – your first chat application

Now, from your MyMQTT mobile app or mosquitto client tool, subscribe to the topic *'chatRoom/<your-name>'*. For e.g., if you've entered the name **'Lara Croft'** in *'client.py'* GUI, then subscribe to the topic *'chatRoom/Lara Croft'*. Type any message in your chat application and press the send button. You'll receive the message in your mosquitto tool or mobile app. Similarly, to send messages from mosquitto tool or mobile app, publish your message to the topic *'chatRoom'* which you'll receive in your chat application. Also, change the QoS value in *'client.py'* file to 0, 1 or 2 and find out how fast the messages are transmitted.

But our chat app still uses the black screened command prompt to receive the message? Moreover, shouldn't our chat application include a game?

So, let us implement a simple Hangman game in a chat room GUI environment. Double-click the *'Hangman.py'* file and you'll see a GUI environment as in Fig. 4.3.

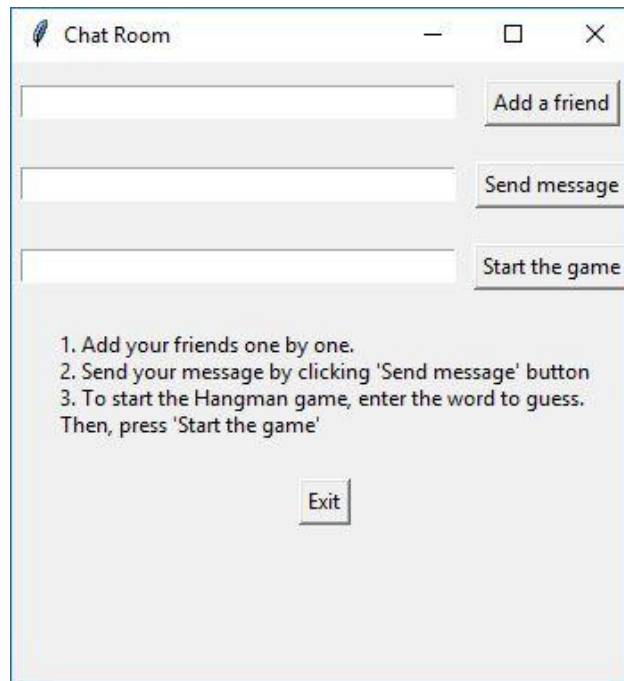


Fig. 4.3 Chat Room window

FYI

We are not going to implement switching lights ON or OFF (as stated in Chapter 1) using MQTT as it requires hardware connection which is beyond the scope of this tutorial. However, you can do it by publishing the messages to a WiFi controlled relay, connected to a lamp, which can switch the lights based on the published messages.



We'll assume that there is only one person who operates our Chat Room application and starts the game. This person, the master, is any person from your MQQM who can give a word for the Hangman game.

To test the app, open your *'client.py'* file multiple times in the same or different computer and enter different names for each window. Add these names in the **'Add a friend'** field in your *'Hangman.py'* window. Send any message you want from the Chat Room and all the client chat applications will receive that message. Similarly, send messages from your client windows too. You can play the hangman game by entering any word like **'pizza'** and pressing **'Start the game'** button. The client chat applications will have to guess the word by sending the character one by one in their chat application (e.g. **'a'**, **'z'** or **'k'**)

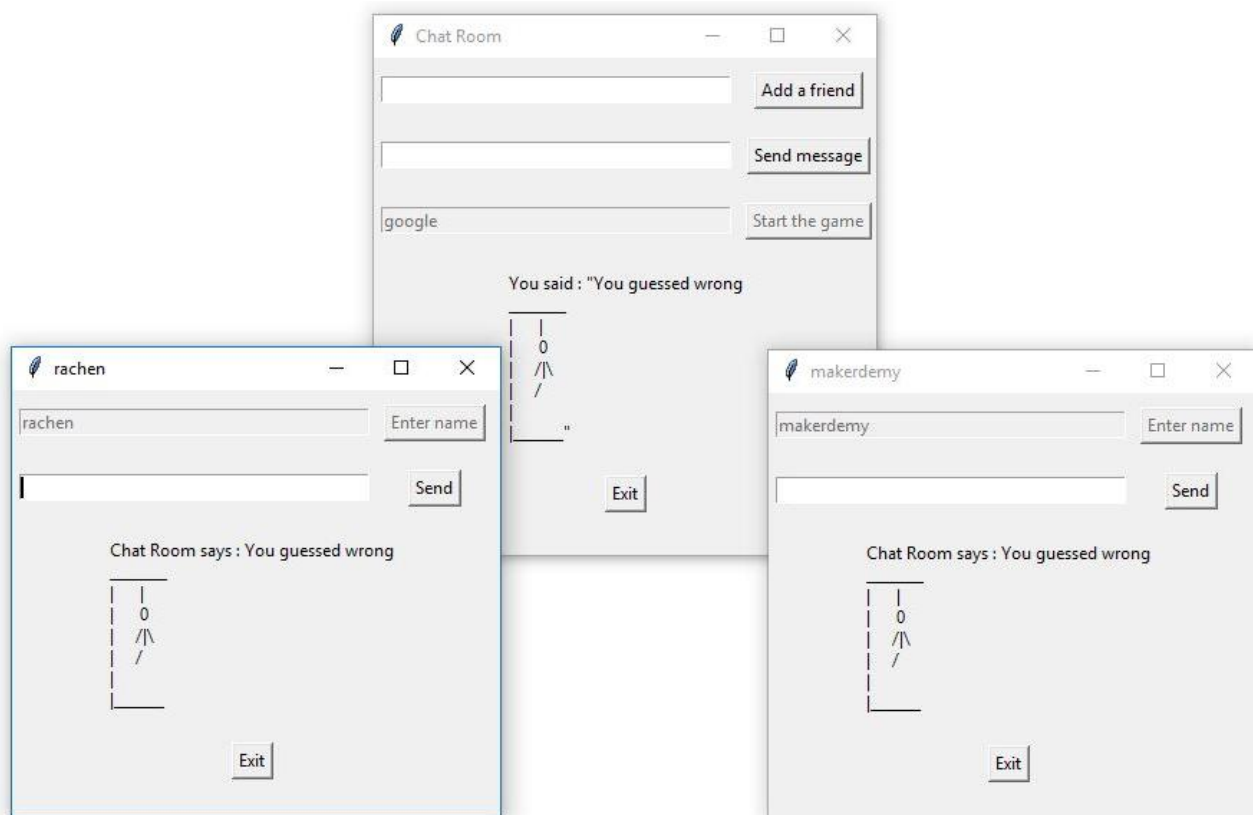


Fig. 4.4. Guessing wrong in Hangman game

This is implemented as python code in *'hangman.py'* file. Right-click the file and select **'Edit with Python IDLE'** to read the code. The explanation for the code is given in comment lines and is actually similar to the *'client.py'* file. Try experimenting with different values of QoS in your python file.

Instead of using *'client.py'* file, you can also use your mobile app or mosquitto broker by subscribing and publishing to the appropriate topics. The following are the list of topics you can use in your app which are subscribed and published by:

'client.py' file: subscribes to *'chatRoom'* and publishes to *'chatRoom/<your-name>'*

'hangman.py' file: publishes to *'chatRoom'* and subscribes to *'chatRoom/<friends'-names>'*

Your app is now ready to be launched. The employers at MQQM are extremely impressed as you have completed the job quickly and with quality. Your colleagues are never bored with the game. If it had not been for MQTT, your chat application would have been dire slow. You are also confident to work on any IoT projects using MQTT. In fact, your employers have confirmed you as a permanent employee in your dream-come-true company and even awarded you the 'Top Notch IoT Developer of the year' award.

But one fine day, your boss calls you to his office and says...

So, you know all about mosquito?
Good for you.
Now get that mosquito out of my office.



