Programming the Server (IoT Part 3 of 3)

In this third part of my IoT series, you will program a Python script to act as a WebSocket server for a collection of microcontrollers. This server will be able to read data from the microcontrollers, and send data to them as well.

1 Download and Run the Code

You can get the server's code from its GitHub Page. Once downloaded, you can run the script by executing the following in a command prompt:

python.exe C:\...full path to server..\PyWsServer.py

If python.exe is NOT in your PATH variable, then just put the full path to python.exe as well. Once executed, the server's GUI should load, and it will look like Figure 1.

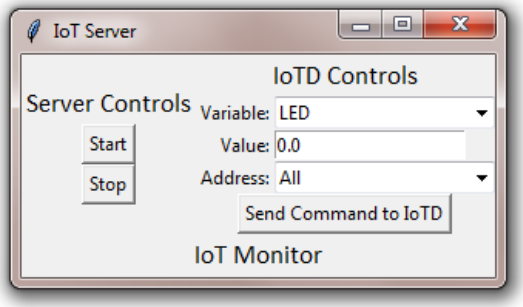

Figure 1 – Python WebSocket Server GUI after startup.

Press the 'Start' button to start the WebSocket server. The microcontrollers will start to connect to the server and stream data. The GUI will expand to start displaying the data as it comes in. An example of a server with five microcontrollers connected is shown in Figure 2.

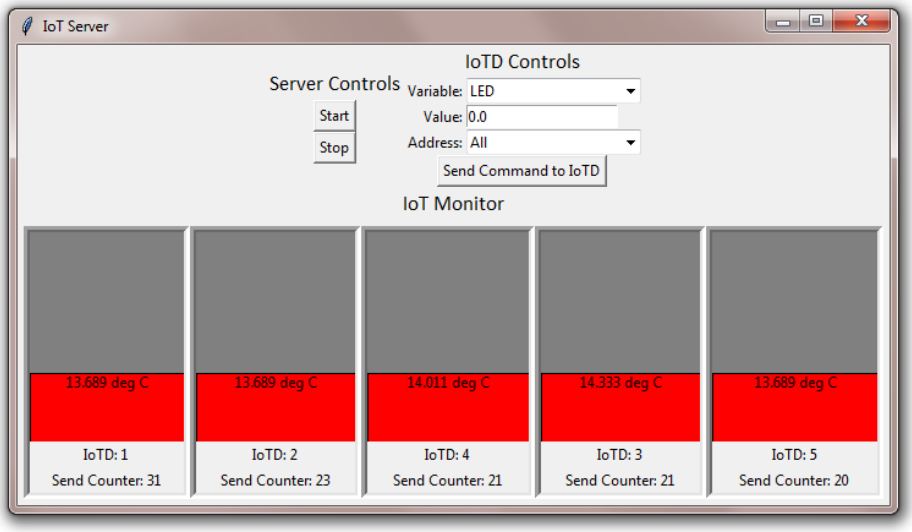

Figure 2 – Python WebSocket Server GUI with 5 IoT Devices connected.

2 Server Controls

There is not too much to the server, but here is a summary of what everything does:

Control	Description
'Start' button	Starts the WebSocket server.
'Stop' button	Stops the WebSocket server and writes memory to disk.
'Variable' combo box	Selects which variable to modify on the microcontroller side. There is currently only a single variable available: LED.
'Value' combo box	Value of the variable to send to the microcontroller. This must be a number.
'Address' combo box	Address of the microcontroller to send data to. When set to 'All', all known microcontrollers will receive the 'Value' for the selected 'Variable'.
'Send Command to IoTD'	Send the data to the microcontroller(s).
'IoT Monitor'	Graphical display of the data coming from the microcontrollers

3 Tour of the Code

As we did with the microcontroller code, we are going to highlight some of the interesting parts of the Server's code.

The equivalent of the microcontroller's infinite main loop is found in PyWsServer.py:

```
170  MyThread = TornadoThread()
171  WebSocketGui = gui.WSGui(MyThread)
172
173  if __name__ == "__main__":
174      # Start the GUI:
175      WebSocketGui.start()
```

All the 'main' loop is doing is calling the gui to start. This starts the gui's 'infinite loop' which keeps the program running. The gui uses the tkinter package, and is defined in gui.py. Unfortunately both the tkinter package and the Tornado package (Recall that the Tonado package is running our WebSocket server) both have their own 'infinite loops' which need to run. In a single threaded program, that means that only one of them could execute at a time. Therefore, the Tornado server is actually running on a different thread than the gui's thread using the threading package. A word of caution: according to Tornado's documentation, the Tornado package is not thread safe. Therefore it should not really be used in this manner. However, since my server is fairly simple, it works just fine.

Once the server has started, it waits for a new WebSocket connection. When one occurs, the WSHandler class is invoked. In particular Line 80 of PyWsServer.py:

```
79   class WSHandler(tornado.websocket.WebSocketHandler):
80       def open(self):
81           info_msg('New connection.')
82           Devices.append(mbedWSClient.MbedWSClient(self))
83           self.index = len(Devices) - 1
84
85       def on_message(self, message):
```

```
86              # Saving message:
87              debug_msg("Received at index: %d message: %s" % (self.index, message))
88              Devices[self.index].append_data(message, IoTDVisualFrame[0])
89              # self.write_message(message[::-1])
90
91          def send_message(self, message):
92              # Send a message to the client:
93              debug_msg("sending message: %s" % message)
94              self.write_message(message)
95
96          def on_close(self):
97              info_msg('connection closed')
98
99          def check_origin(self, origin):
100             return True
101
102         def is_wsconnected(self):
103             if self.ws_connection is None:
104                 return False
105             else:
106                 return True
```

When a new connection is opened, it is saved in the Devices[] array as an MbedWSClient class (Line 82). This class is defined in mbedWSClient.py. The Devices[] array handles all of the connections, meaning the size of this array could be larger than the number of devices actually connected to the server, since every new connection will invoke a new instance of the MbedWSClient class. The other methods in the WSHandler class are fairly self-explanatory, but most rely on methods in the MbedWSClient class in mbedWSClient.py, which we will take a look at now:

```
83  class MbedWSClient(object):
84      frameDict = {}
85      clientDict = {}
86      lastSaveDict = {}
87      lastSaveDate = {}
88
89      def __init__(self, wshandle):
...
93      def append_data(self, data_string, iot_frame):
...
126     def save_data_to_disk(self, iot_to_save):
...
189     def send_command(self, cmd, value):
...
192     def send_cmdstr(self, cmd):
...
195     def get_id(self):
...
198     def is_connected(self):
```

For the most part, the method names give a clear indication of what each of them do. The variables defined on Lines 84 to 87 are variables common to all instances of the MbedWSClient class. Recall that each instance of an MbedWSClient class corresponds to a WebSocket connection. Each physical microcontroller, or IoT device, corresponds to a single entry in each one of the dictionaries of Lines 84 to

87. The ID of the IoT devices serves as the key for each dictionary. Each dictionary holds different information:

Dictionary	Description
frameDict{}	A dictionary of IoTVisual objects, one for each IoTD. The IoTVisual class handles the graphical representation of each microcontroller's data in the gui.
clientDict{}	A dictionary of MbedData objects, one for each IoTD, or microcontroller. The MbedData class handles all of the data coming from the microcontroller.
lastSaveDict{}	A dictionary of integers, one for each IoTD, or microcontroller. This integer represents the last index of the MbedData that has been saved to disk.
lastSaveDate{}	A dictionary of date, one for each IoTD, or microcontroller. This date represents the last date which was saved to disk. The data saved to disk is separated by date and this variable ensures that only data form the same day is saved in the same file.

If you want to modify the data that is being send or received by the server from each microcontroller, take a closer look at the `append_data` method of Line 93, and the `save_data_to_disk` method of Line 126. A quick way to add more variables however is to simply add their indices, and change the value of MAXVALUES at the start of mbedWSClient.py:

```
62    SENDCOUNTERIDX = 1
63    TEMPIDX = 2
64    MAXVALUES = 3
```

Recall that the microcontrollers for this example are sending data in the form of:

```
IoT_ID, SendCounter, TempSensor
```

Therefore if you want to add a third piece of data to send, the microcontrollers could send:

```
IoT_ID, SendCounter, TempSensor, Gain
```

and you would just have to modify mbedWSClient.py to:

```
62    SENDCOUNTERIDX = 1
63    TEMPIDX = 2
64    GAINIDX = 3
65    MAXVALUES = 4
```

4 Summary

At this point you should this IoT example up and running. It is a very basic, but reliable wireless communication link between microcontrollers and a server. Once the data reaches the Server, you can disseminate it to other computers over the internet. Furthermore, with the correct firewall settings on your router, the microcontrollers could be anywhere in the world, feeding data back to your home server.

This example is just the tip of the iceberg in terms what is possible with low-cost microcontroller hardware. Many improvements could be made to the code, but I hope this has been a valuable first step for your own IoT project!