Socket programming on Raspberry Pi

Microprocessor Systems and Applications

GITHUB.COM/CLAUDIUGROZA/MSA

February 2018

Contents

1. Socket programming

The purpose of this guide is to offer an implementation walkthrough of a client-server application on Raspberry Pi. The next example will control an LED placed on the server side. The state of the LED is controlled by sending a message from the client side which contains the state's value to be set.
In order to reduce the level of verbosity, we assume all required packages are imported in our scripts. Furthermore, we concentrate only on the key sections of the script. Don't worry about that too much because the full versions of the scripts are available via the repository.

1.1 Server side

We start with the part which defines the parameters of the TCP/IP connection [Soc]:

```
# provide address as a tuple
server_address = ('localhost', 1221)

# create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# associate socket with server address
sock.bind(server_address)

# start listening for incoming connections
sock.listen(1)
```

Next step is to wait for an arbitrary client to initiate a connection:

```
connection, client_address = sock.accept()
```

Immediately after accepting the connection from a client, we are able to receive data:

```
buff = b''
try:
    while True:
        # receive data in chunks
        data = connection.recv(8)
        if data:
            buff += data
        else:
            break

    # parse received data
    json_recv = json.loads(buff.decode('utf-8'))

    # get state's value
    state = json_recv['state']

    # set LED accordingly
    set_output(pin, state)

finally:
      connection.close()
```

Once the transmission of data is successfully completed, we can proceed to extract the new state value. Firstly, the plain binary format should be converted to a JSON format [Jso]. Afterwards, we extract the state's value from the JSON object and pass it to a function that has the sole responsibility to control the state on an LED.

Let's now run the script we already discussed:

```
$ sudo python server.py
```

Because we have not finished yet the script which behaves as a client side, we need a tool that enables us to send TCP messages to an address. **Netcat** network utility [Net] will be used to help us send some sample messages:

```
$ echo '{"state":0}' | nc localhost 1221
```

The output of the *server* script should almost resemble the following text:

```
waiting connection
13:54:20 connected to ('127.0.0.1', 55247)
setting "0" to pin 12
13:54:20 disconnected from ('127.0.0.1', 55247)
```

1.2 Client side

We assume that until now you have a grasp of basic socket programming in Python. The next section discusses the part that sends a message to the server we built in the previous section.

As presented earlier, we start by instantiating a socket object:

```
server_address = ('localhost', 1221)
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

From this moment things are starting to change. We do not listen for an incoming connection, but we initiate a connection via the defined socket object:

```
# start a connection
sock.connect(server_address)
```

Once a connection was established, we compose the message to be transmitted and then send it over the socket using the JSON format:

```
try:
    # build dict to send
    value = {'state' : state}

    # encode object
    encoded_object = json.dumps(value)

    # send data
    sock.sendall(encoded_object)

finally:
      sock.close()
```

To test the client side we can use either the server script or the handy *netcat*'s socket listening functionality. The later is our choice and we continue by typing the following command to open a socket which listens for incoming connections on the **local** machine and port **1221**:

```
$ nc -l 1221
```

The last step would be to run the client script:

```
$ sudo python client.py
```

The output should be similar with the following text:

```
11:31:42 sent {"state": 0}
closing socket
```

2. Assignments

1. Extend *client.py* script such that listens for button events. Once the button was pressed, the client side sends a request to revert the current state of the LED. In order to reduce complexity, we assume that the current state of LED is known by the client.

2. Extend *server.py* script to support the read operation on a port. Your server implementation must distinguish between READ and WRITE messages. The write logic is already handled, so you need to concentrate on how to read the port's state. In both of these cases the server should return the port state back to its client. Use netcat tool to compose and send messages to the server. A message format example you can use is presented next:

   READ
   { "operation" : "read", "pin": 12}

   WRITE
   { "operation": "write", "pin": 12, "state": 1 }

   READ/WRITE **response**
   { "pin": 12, "state": 1 }

3. Extend *server.py* script such that clients can configure the direction of a specific port (input or output).
   Your implementation should also handle error cases. For example, if a port is not yet configured but a client tries to read from that port, the server must return an error message. Take a look on the following message formats:

CONFIGURE
{ "operation": "config", "direction": "input" , "pin": 10}

ERROR response
{ "error": 1, "message": "Not yet configured" }

3. Bibliography

3.1 References

[Jso] *JSON docs*. http://docs.python-guide.org/en/latest/scenarios/json. [Online; accessed March-2018]. 2018 (cited on page 6).

[Net] *Netcat docs*. http://www.tutorialspoint.com/unix_commands/nc.htm. [Online; accessed March-2018]. 2018 (cited on page 6).

[Soc] *Socket networking*. https://docs.python.org/2/howto/sockets.html. [Online; accessed March-2018]. 2018 (cited on page 5).

3.2 Image credits

- First page illustration.
  http://www.northeastern.edu/levelblog/2018/01/25/guide-iot-careers
- Chapter header background.
  https://blogs.microsoft.com/iot/2015/03/17/simplifying-iot/