Back to tutorial index

Python Web-based Serial Console using WebSockets

A good approach for creating user interfaces to serial devices is to build an interactive web page. Using a web-based GUI makes the interface portable to different screens and devices. A micro computer such a Raspberry PI can then act as both a controller for the device and as webserver for providing the interface.

This tutorial shows how to do this using Python and Web Sockets.

Requirements and Architecture

Before digging into the details let's describe the requirements:

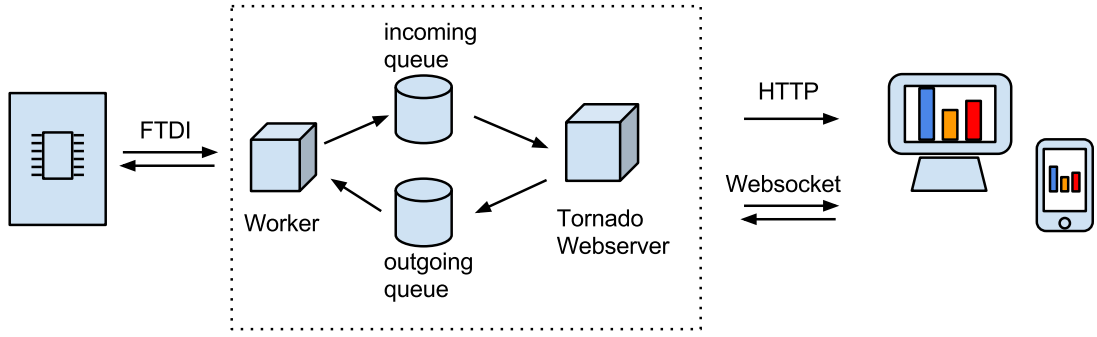
we have a device communicating via serial protocol to a computer, often using an FTDI interface

we want to send data to and read data from the serial port, this could be commands or raw data coming from sensors

we want to manipulate this data in a HTML webpage using Javascript

we want the communication not to happen by "polling", i.e. querying the webserver for changes every once in a while, but by keeping an always-open communication channel from the webpage to the webserver, using the Web Sockets standard

So this is our architecture:



Architecture diagram

The device will read and write to the serial port in an infinite loop.

The Webserver will have a separate "worker" thread dealing with the serial port, using an incoming and outgoing queue to handle concurrency.

Every time there's a message in the incoming queue, the webserver main thread will write it "emit it" to the websocket.

Every time there's a message in the outgoing queue, the "worker" thread will write it to the serial port.

Within the web browser, the communication will be event-based. Every time a new message will be received from the websocket, the GUI will be updated. Every time the user will send a command, i.e. pushing a button, a command message will be sent on the websocket.

Required software

We will need to install several python package. The tutorial assumes you're using Python 2.7.x and we can install packages via pip:

```
sudo pip install pyserial tornado multiprocessing
```

The pyserial module will allow us to access the serial port in the python environment. The tornado module contains all code needed to serve webpages and create websockets. The multiprocessing modules has the features needed to write concurrent code in python.

Web page

We can code a simple HTML5 webpage that will host our user interface, but also send and receive data using the websocket.

```
<!DOCTYPE HTML>
<html>
  <head>
    <style>
      body { margin: 0px; padding: 20px; }
      #received { width: 500px; height: 400px; border: 1px solid #dedede; overflow-y:scroll;}
      #sent { width: 500px; }
    </style>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
    <script type="text/javascript" src="static/main.js"></script>
```

```
  </head>
  <body>
  <h1>Websockets serial console</h1>

  <p>Data received from serial port</p>
  <div id="received">
  </div>
  <button id="clear">Clear</button>

  <p>Send data to serial port</p>
  <form id="sent">
    <input type="text" id="cmd_value">
    <button id="cmd_send">Send</button>
  </form>
  </body>
</html>
```

The Javascript code is contained in the main.js javascript below. We use the browser WebSocket object for interacting with the /ws endpoint.

```
$(document).ready(function(){

    var received = $('#received');


    var socket = new WebSocket("ws://localhost:8080/ws");

    socket.onopen = function(){
      console.log("connected");
    };

    socket.onmessage = function (message) {
      console.log("receiving: " + message.data);
      received.append(message.data);
      received.append($('<br/>'));
    };

    socket.onclose = function(){
      console.log("disconnected");
    };

    var sendMessage = function(message) {
      console.log("sending:" + message.data);
      socket.send(message.data);
    };


    // GUI Stuff


    // send a command to the serial port
    $("#cmd_send").click(function(ev){
      ev.preventDefault();
      var cmd = $('#cmd_value').val();
      sendMessage({'data' : cmd});
      $('#cmd_value').val("");
    });

    $('#clear').click(function(){
      received.empty();
    });


});
```

Tornado web server

The tornado web server allows us to serve webpages and also communicate via websockets.

```
import tornado.httpserver
import tornado.ioloop
import tornado.web
import tornado.websocket
import tornado.gen
from tornado.options import define, options

import time
import multiprocessing
import serialProcess
```

```
define("port", default=8080, help="run on the given port", type=int)

clients = []

class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        self.render('index.html')

class WebSocketHandler(tornado.websocket.WebSocketHandler):
    def open(self):
        print 'new connection'
        clients.append(self)
        self.write_message("connected")

    def on_message(self, message):
        print 'tornado received from client: %s' % message
        self.write_message('got it!')

    def on_close(self):
        print 'connection closed'
        clients.remove(self)

if __name__ == '__main__':
    tornado.options.parse_command_line()
    app = tornado.web.Application(
        handlers=[
            (r"/", IndexHandler),
            (r"/ws", WebSocketHandler)
        ]
    )
    httpServer = tornado.httpserver.HTTPServer(app)
    httpServer.listen(options.port)
    print "Listening on port:", options.port
    mainLoop = tornado.ioloop.IOLoop.instance()
    mainLoop.start()
```

In the code we keep track of connected clients using the "clients" list, then we start listening on the http port (default is 8080) for standard http requests on the "/" path, and for websocket requests on the '/ws' path.

Serial worker

Now let's move to the serial worker. This background task will be interacting with the serial device, by reading data from the serial port and posting it to the incoming queue, and by reading data from the outgoing queue and writing it to the serial port.

As you see in the code below the SerialWorker class extends the multiprocessing package "Process" class, that contains all the code to create a background daemon. We only need to define a run() method which will implement our read-write loop.

When instancing the SerialProcess you need to pass in the input and output queue as parameters.

```
import serial
import time
import multiprocessing

## Change this to match your local settings
SERIAL_PORT = '/dev/ttyACM0'
SERIAL_BAUDRATE = 115200

class SerialProcess(multiprocessing.Process):

    def __init__(self, input_queue, output_queue):
        multiprocessing.Process.__init__(self)
        self.input_queue = input_queue
        self.output_queue = output_queue
        self.sp = serial.Serial(SERIAL_PORT, SERIAL_BAUDRATE, timeout=1)

    def close(self):
        self.sp.close()

    def writeSerial(self, data):
        self.sp.write(data)
        # time.sleep(1)

    def readSerial(self):
        return self.sp.readline().replace("\n", "")

    def run(self):

        self.sp.flushInput()

        while True:
            # look for incoming tornado request
```

```
            if not self.input_queue.empty():
                data = self.input_queue.get()

                # send it to the serial device
                self.writeSerial(data)
                print "writing to serial: " + data

            # look for incoming serial data
            if (self.sp.inWaiting() > 0):
                data = self.readSerial()
                print "reading from serial: " + data
                # send it back to tornado
                self.output_queue.put(data)
```

Putting it all together

As mentioned in the previous sections the key for having i/o on the serial port without blocking the webserver is creating two separate threads running in parallel and communicating via shared queues.

The Tornado webserver has his own thread, that we start using the tornado.ioloop.IOLoop class start method. No further instruction will be executed after we start this loop, unless we interrupt the application.

So we need to start the Serial Worker thread before the Web server. Then we should add a way for the Web Server to periodically check what should be posted to the websocket, and what has been posted on it by the clients.

In order to check the incoming queue, we can make use of the tornado.ioloop.PeriodicCallback scheduler, which schedules a given callback to be called periodically.

For the outgoing queue, we can simply write to it each time we receive a message from the websockets. With all of this in mind the server.py file will look like:

```
import tornado.httpserver
import tornado.ioloop
import tornado.web
import tornado.websocket
import tornado.gen
from tornado.options import define, options
import os
import time
import multiprocessing
import serialworker
import json

define("port", default=8080, help="run on the given port", type=int)

clients = []

input_queue = multiprocessing.Queue()
output_queue = multiprocessing.Queue()



class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        self.render('index.html')

class StaticFileHandler(tornado.web.RequestHandler):
    def get(self):
        self.render('main.js')

class WebSocketHandler(tornado.websocket.WebSocketHandler):
    def open(self):
        print 'new connection'
        clients.append(self)
        self.write_message("connected")

    def on_message(self, message):
        print 'tornado received from client: %s' % json.dumps(message)
        #self.write_message('ack')
        input_queue.put(message)

    def on_close(self):
        print 'connection closed'
        clients.remove(self)


## check the queue for pending messages, and rely that to all connected clients
def checkQueue():
    if not output_queue.empty():
        message = output_queue.get()
```

```
        for c in clients:
            c.write_message(message)


if __name__ == '__main__':
    ## start the serial worker in background (as a deamon)
    sp = serialworker.SerialProcess(input_queue, output_queue)
    sp.daemon = True
    sp.start()
    tornado.options.parse_command_line()
    app = tornado.web.Application(
        handlers=[
            (r"/", IndexHandler),
            (r"/static/(.*)", tornado.web.StaticFileHandler, {'path': './'}),
            (r"/ws", WebSocketHandler)
        ]
    )
    httpServer = tornado.httpserver.HTTPServer(app)
    httpServer.listen(options.port)
    print "Listening on port:", options.port

    mainLoop = tornado.ioloop.IOLoop.instance()
    ## adjust the scheduler_interval according to the frames sent by the serial port
    scheduler_interval = 100
    scheduler = tornado.ioloop.PeriodicCallback(checkQueue, scheduler_interval, io_loop = mainLoop)
    scheduler.start()
    mainLoop.start()
```

Source files

WebSocketConsole.zip