Control System User Manual

V1.1

Winter 2024

Content

1.0 Preface

This document stands as a guide to working with broadcast television equipment in a control system. It is important to note that as the broadcast environment grows and becomes more closely entwined with traditional and developing IT systems, this document will drift into wider IT control.

Certain parts of this document will have equipment specific addressing redacted or altered to reduce confusion or reference to any specific equipment. The intent is for this guide to be as universal as possible.

2.0 Flask Web Server

Flask is a micro web framework, written in the Python programming language. It is classed as micro as it has a very little footprint that is built up by the user, depending on what type of use is intended. We use Flask over similar frameworks like Django for this reason. Our requirements from web server interacting with routers etc are quite minimal and require a small amount of imports/ libraries.

We will discuss this in more detail later in the section.

2.1 Flask Overview

As stated above, Flask is a small framework, with a very minimal footprint.
Out of the box, once Flask is imported into your Python script, flask will offer you the ability to access a render template, normally some html text, from the local host on a specified port.

The focus is on a low footprint, so you get nothing else. If you require the ability to interact with the OS, make http requests, handle json information, you will need to install and import these libraries into the script separately.

Flask is typically installed using Pythons pip repository. Installation instructions are found below.

Once Flask is imported into your Python script, the script itself acts as the backend to the web server, so all functions are written in the same script, typically, this is actually up to you and how you choose to write your web servers.

2.2 Installing Flask

Prerequisites: An up-to-date python installation. Flask is supported in Python 3.8 and beyond.

Install instructions;

```
pip install Flask
```

... Well done, Flask installed on your server.

Note, the following packages will be installed on the server as dependencies automatically;

Werkzeug
Jinja
MarkupSafe
ItsDangerous
Click
Blinker

It is up to you to decide if you are happy running these services on the particular server hosting the Flask framework.

Once this step is done, you are ready to start building Flask web servers.

Here is an example of a very basic web server;

Python

```python
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"


if __name__ == '__main__':
    app.run(debug=True, host="0.0.0.0", port=5000)
```

The above code snippet is a complete web server written in Flask, this is what is meant by low footprint. The webserver above only displays the text "Hello World", which on it's own is not useful, but from here you can build applications that make routes on Park Royal A/V Routers and display current system information for end users.

Running the application is as easy as running the following line of code in a terminal

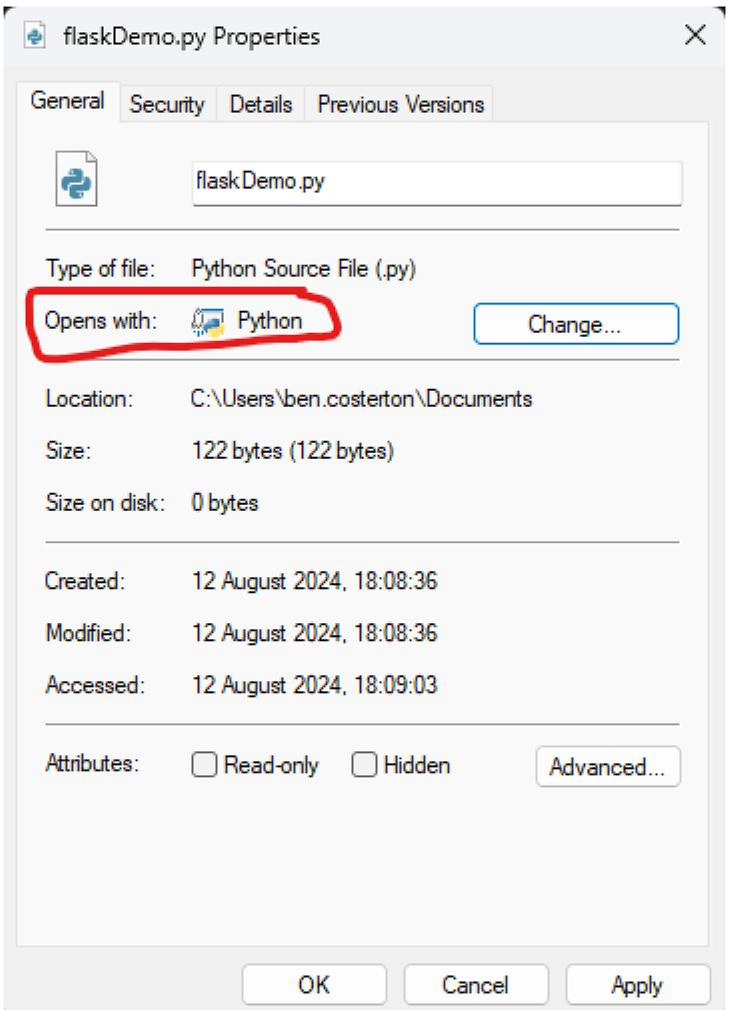
```
$ python flaskDemo.py
```

Which will display the following in the terminal output;

```
 * Serving Flask app 'flaskDemo'
 * Debug mode: on
```

```
WARNING: This is a development server. Do not use it in a production deployment. Use a
production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://xx.xx.xx.xx:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 236-362-646
```

Note: using this method of 'python flaskDemo.py (Where "flaskDemo.py" is the name of the python file containing the above script), will require python to be the default application for the .py scripts on your machine. In windows this can be set as below.



Explaining the terminal output on server launch;

```
 * Serving Flask app 'flaskDemo'
```

Tells us the file used to launch the Flask application.

```
* Debug mode: on
```

When debug mode is on, we see terminal output when the server is interacted with, this is what we want.

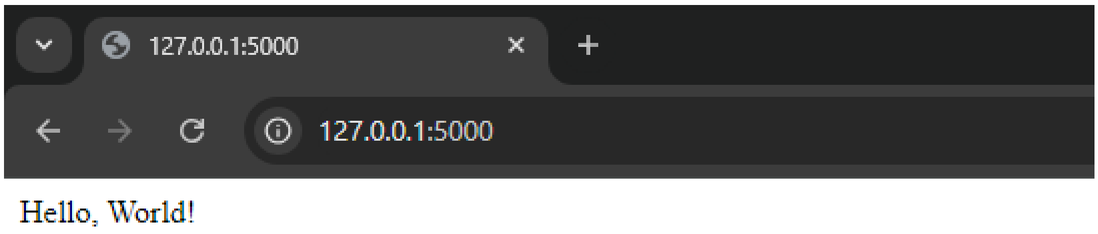
```
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
```

The above warning highlights the fact that there is no Web Server Gateway Interface in use. As a result, the web server will only be able to handle very basic requests. For example, the server will only be capable of processing one request at a time. This could be a request for a html template, a http request to a server location, a button press by a user.
This is a point of consideration when designing, but as we are not launching applications to be used by many people at once, this is not an issue for us. If the server will be used by one set of users to change router settings, or display information, we can cope without a WSGI.

```
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://xx.xx.xx.xx:5000
```

These lines here tell us where the server is accessible from. As we specified a host and port in the code "`host="0.0.0.0", port=5000)`", the above information simply confirms what we wrote. The server can be accessed from local port 127.0.0.1:5000, and from our LAN address xx.xx.xx.xx:5000, which will be used by users or other control interfaces, Cerebrum for example, when accessing this server.

So what does this example server look like to end users?



That's pretty rad!

If you've got this far, rest assured Flask is installed on the server, and you are ready to start hosting web applications for end users. From here out, it's configuration as the application requires.

2.3 Backend (Python)

As we have seen above, Flask servers are written in Python.
The following will not be a tutorial in using Python or Python best practices, but hopefully will highlight some of the important syntax for building Flask servers in Python and trigger events from a front end script.

Folder structure

Before we get into functions and operations, we need to talk about Flask folder structure, as it is slightly rigid.
Below is an example of the folder structure used for Flask web servers;

```
MyFirstFlaskServer/
├── app.py    <--- This file is the Flask web server
├── templates/
│    └── index.html
├── static/
│    ├── logo.jpg
│    └── styles.css
└── xml/
     ├── Command_1.xml
     └── Command_2.xml
```

Your Python script will live in the root of the directory, always.

There must be a folder titled 'templates', this is where your html files will live.
If you wish to style your control page using css or storing images, you will need a 'static' folder.
Any additional requirements, for example storing xml commands, should be kept in a separate folder, and specified in the script.

@app.route

The @app.route is one of the very basic functions you will come across when writing Flask servers. In short, the app.route maps urls to a specific function in your script. When a user navigates to a url '/' for example, or '/button001' a specific function in your script will executed.

A common use for this is to map a button press on your front end to navigate to a url. A example of this written in html;

HTML

```html
<button id="bol1_g1_floor" class="button" type="button" onclick="buttonClicked(this);"
data-endpoint="/bol1_g1_floor">G1 Floor</button>

<script>
    function buttonClicked(button) {
    var buttons = button.parentElement.querySelectorAll('.button'); // Select buttons within the
same group
    disableButtons(buttons); // Pass the buttons to disableButtons function

    var endpoint = button.getAttribute('data-endpoint');
    // This is the bit that actually sends the command to the router
    var xhr = new XMLHttpRequest();
    xhr.open('POST', endpoint);
    xhr.onreadystatechange = function() {
```

```
        if (xhr.readyState === XMLHttpRequest.DONE) {
            setTimeout(function() {
                enableButtons(buttons); // Pass the buttons to enableButtons function
            }, 2000); // Adjust the delay time as needed
        }
    };
    xhr.send();
</script>
```

In the above example, we have a '<button>' element "bol1_g1_floor".
This will render a button on a webpage for a user to press.

Once the button is pressed, the actions outlined in the '<button>' element will be action. In this example it triggers the endpoint "/bol1_g1_floor".

The function bellow, written in JavaScript between the '<script>' identifiers actions sends this endpoint command to the backend server (our Python script).

There is some extra script found above to disable the button once pressed, this is just to prevent multiple requests being made to the backend in quick succession.

When the above button is pressed, the endpoint "/bol_g1_floor" is sent to the backend server (our python script), and will look something like this;

Python

```python
@app.route('/bol1_g1_floor', methods=['POST', 'GET'])
def bol1_g1_floor():
    bol1_key2_park()
    script_dir = os.path.dirname(os.path.abspath(__file__))
    xml_files = ['bol1_key2_fm1.xml']
    server_address = 'http://xx.xx.xx.xx:8193'
    headers = {'Content-Type': 'text/xml'}
    for xml_file in xml_files:
        xml_file_path = os.path.join(script_dir, 'xml', xml_file)
        with open(xml_file_path, 'r') as f:
            xml_data = f.read()
        requests.post(server_address, data=xml_data, headers=headers)
    return ' '
```

Above is the @app.route that is linked to the html text we saw earlier.
Immediately after the @app.route we see the end point specified, in the example `@app.route('/bol1_g1_floor'`. This states that when a user request the endpoint "/bol1_g1_floor", the following code should be executed.

We can also specify the methods in which this code is executed, we will cover this below in *requests*.

render_template

render_template is a function normally used to serve a webpage to the user on accessing the server. For example, if a user was to access the server via a browser, using out LAN address of http://xx.xx.xx.xx:5000 (remember this is user specified in the code, see above) the endpoint '/' will be requested from the server.

'/' Is used when the user does not specify any particular end point, so this is where we will put our default landing page, or control interface etc.

The function in Python will look something like this;

Python

```python
@app.route('/', methods=['GET', 'POST'])
def index():
    return render_template('boleroconfig.html')
```

When a user navigates to our server address, the a 'return' function is used to provide the html file 'boleroconfig.html' using the render_template function.

All render templates must live in the 'templates' folder as specified above. This is where the 'render_template' function will go and look for them.

You can use multiple .html files of course, in your code it will look like this;

Python

```python
@app.route('/', methods=['GET', 'POST'])
def index():
    return render_template('boleroconfig.html')

@app.route('/bol1_config')
def bol1_config():
    return render_template('bol1_config.html')

@app.route('/bol2_config')
def bol2_config():
    return render_template('bol2_config.html')
```

In the above example, when the user accesses the server initially, they are provided the boleroconfig.html file, and if the user needs to access a specific page on the control server, they will be provided a specific .html file, for example 'bol1_config.html'.

requests [GET, POST]

The request function is not a standard Python library, and will need to be installed;

```
python pip install requests
```

The request module is used to handle HTTP requests within your script. As is common with control servers, we will need to communicate with routers and switches, and http requests is common method of communication.

Two common methods of request functions are 'GET' and 'POST'.

The GET request is used to request information from a server. For example, if you need to retrieve information from a server, in order to process that data and present it to the user, you will use the GET function.

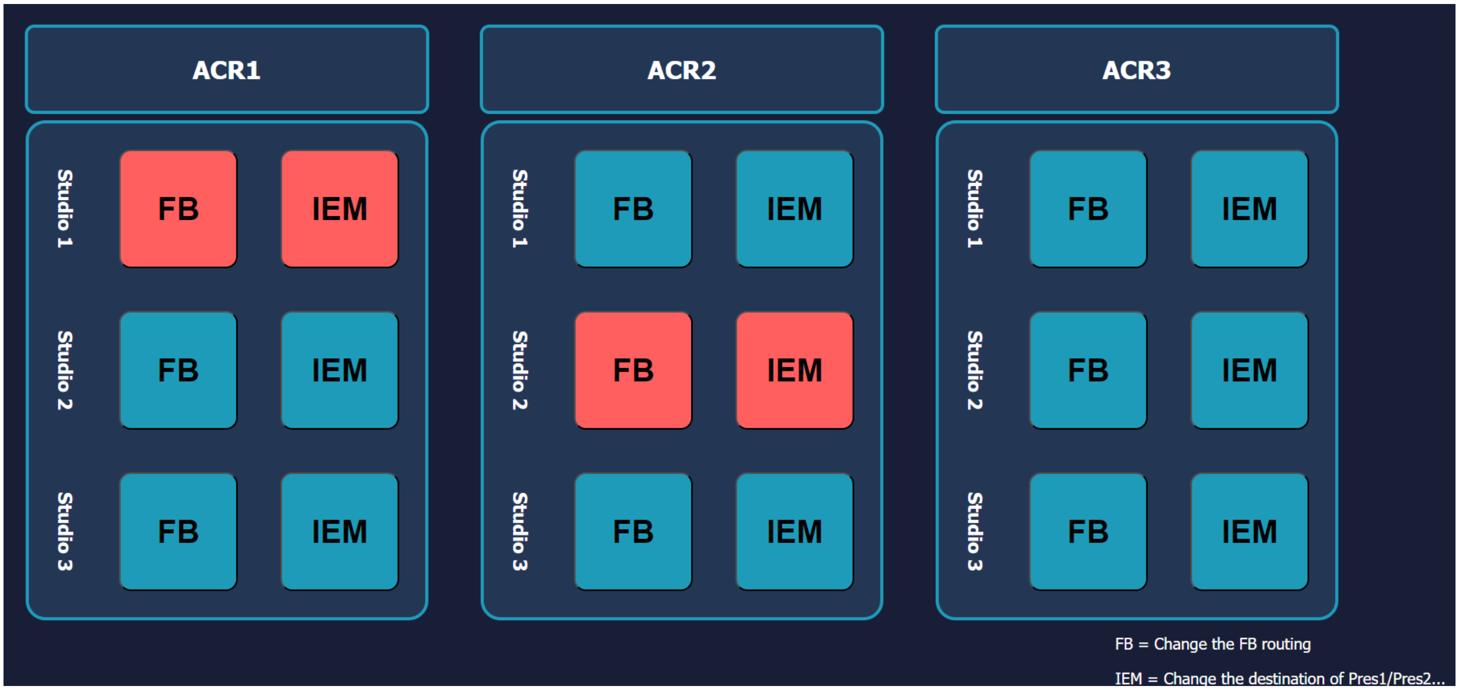
The POST method is used to send information, including a payload, to a server to update that remote server in some way. This is commonly used in control systems to send router commands to a router, for example.

jsonify

jsonify is a library used to format information gathered from a server in a JSON.
This is very useful when providing a list of information to a html function that will update button values, for example.

If we take a control interface used make routes on a router, it is a necessity to know what routes are currently active.

Take the Foldback and IEM routing page as an example;



The above page not only allows the user to make/break routes, but also highlights the routes currently active on the system.

Together, this page hosts statuses of 18 different crosspoint on the router, so it is helpful to format this in some way. JSON is a good choice for this.

If we use the FB routing as an example, we find the following function in our html file;

HTML

```
function get_crosspoint_statuses() {
var xhr = new XMLHttpRequest();
xhr.open('POST', '/get_crosspoint_statuses');
xhr.send();

xhr.onload = function() {
  if (xhr.status === 200) {
      // Parse the JSON response from the server
      var response = JSON.parse(xhr.responseText);

      // Call the function to update the button status
      updateButtonStatuses(response);
  } else {
      alert('Request failed. Status: ' + xhr.status);
  }
};
```

```
}
setInterval(get_crosspoint_statuses, 2000);
```

This function is running in the '<head>' of the html file, and has a setInterval function that will run the function every two seconds. The function sends the endpoint to the server;
`xhr.open('POST', '/get_crosspoint_statuses');`

The flask application has the following function to handle the request;
Python

```python
@app.route('/get_crosspoint_statuses', methods=['POST', 'GET'])

def get_crosspoint_statuses():
    script_dir = os.path.dirname(os.path.abspath(__file__))
    server_address = 'http://127.0.0.1:8193'
    headers = {'Content-Type': 'text/xml'}
    xml_files = {
        'ACR1_FB1_GetXpStatus.xml': 'ACR1_FB1',
        'ACR1_FB2_GetXpStatus.xml': 'ACR1_FB2',
        'ACR1_FB3_GetXpStatus.xml': 'ACR1_FB3',
        'ACR2_FB1_GetXpStatus.xml': 'ACR2_FB1',
        'ACR2_FB2_GetXpStatus.xml': 'ACR2_FB2',
        'ACR2_FB3_GetXpStatus.xml': 'ACR2_FB3',
    }

    crosspoint_statuses = {}
    for xml_file, key in xml_files.items():
        xml_path = os.path.join(script_dir, 'xml', xml_file)

        with open(xml_path, 'r') as f:
            xml_data = f.read()

        try:
            response = requests.post(server_address, data=xml_data, headers=headers)
            response.raise_for_status()
            response_text = response.text
            has_error, message = process_boolean_response(response_text)
            crosspoint_statuses[key] = not has_error

        except requests.exceptions.RequestException as e:
            crosspoint_statuses[key] = False
```

```
    return jsonify(crosspoint_statuses)
```

The output of the response from the server is converted to JSON format in the last line if the function;

```
    return jsonify(crosspoint_statuses)
```

The JavaScript function above in the html block then processed the JSON file;

```
xhr.onload = function() {
  if (xhr.status === 200) {
      // Parse the JSON response from the server
      var response = JSON.parse(xhr.responseText);
```

...and we can see the output of this response in our browser;

```
{
  "ACR1_FB1": true,
  "ACR1_FB2": false,
  "ACR1_FB3": false,
  "ACR2_FB1": false,
  "ACR2_FB2": true,
  "ACR2_FB3": false
}
```

The html file will then be able to use this JSON string to update that color of the buttons to indicate what crosspoints are 'true' or active.

xml.etree.ElementTree (xml api)

The xml.etree.ElementTree is a module used to create and parse xml data.
Xml files are a common form of sending and receiving data from a router, so this module is likely to be highly used.

For example, if you are requesting information from a router, it may reply with a xml file. Bellow is an example a of a response from the Riedel RRCS service replying to a request for all active clone ports on the system;

XML⤵

```
<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
<params>
<param>
  <value><array><data>
    <value><string>C0123456789</string></value>
```

```
    <value><array><data>
      <value><struct>
        <member>
          <name>ClonePort</name>
          <value><struct>
            <member>
              <name>IsInput</name>
              <value><boolean>0</boolean></value>
              </member>
            <member>
              <name>Node</name>
              <value><i4>2</i4></value>
              </member>
            <member>
              <name>Port</name>
              <value><i4>34</i4></value>
              </member>
            </struct></value>
          </member>
        <member>
          <name>PortToMonitor</name>
          <value><struct>
            <member>
              <name>IsInput</name>
              <value><boolean>0</boolean></value>
              </member>
            <member>
              <name>Node</name>
              <value><i4>4</i4></value>
              </member>
            <member>
              <name>Port</name>
              <value><i4>52</i4></value>
              </member>
            </struct></value>
          </member>
        </struct></value>
      <value><struct>
        <member>
          <name>ClonePort</name>
          <value><struct>
            <member>
              <name>IsInput</name>
              <value><boolean>0</boolean></value>
              </member>
            <member>
              <name>Node</name>
              <value><i4>2</i4></value>
              </member>
            <member>
              <name>Port</name>
              <value><i4>35</i4></value>
              </member>
            </struct></value>
          </member>
        <member>
          <name>PortToMonitor</name>
          <value><struct>
```

```
          <member>
            <name>IsInput</name>
            <value><boolean>0</boolean></value>
            </member>
          <member>
            <name>Node</name>
            <value><i4>4</i4></value>
            </member>
          <member>
            <name>Port</name>
            <value><i4>53</i4></value>
            </member>
          </struct></value>
        </member>
      </struct></value>
    <value><struct>
      <member>
        <name>ClonePort</name>
        <value><struct>
          <member>
            <name>IsInput</name>
            <value><boolean>0</boolean></value>
            </member>
          <member>
            <name>Node</name>
            <value><i4>3</i4></value>
            </member>
          <member>
            <name>Port</name>
            <value><i4>54</i4></value>
            </member>
          </struct></value>
        </member>
      <member>
        <name>PortToMonitor</name>
        <value><struct>
          <member>
            <name>IsInput</name>
            <value><boolean>0</boolean></value>
            </member>
          <member>
            <name>Node</name>
            <value><i4>4</i4></value>
            </member>
          <member>
            <name>Port</name>
            <value><i4>54</i4></value>
            </member>
          </struct></value>
        </member>
      </struct></value>
    <value><struct>
      <member>
        <name>ClonePort</name>
        <value><struct>
          <member>
            <name>IsInput</name>
            <value><boolean>0</boolean></value>
```

```xml
          </member>
        <member>
          <name>Node</name>
          <value><i4>3</i4></value>
          </member>
        <member>
          <name>Port</name>
          <value><i4>55</i4></value>
          </member>
        </struct></value>
      </member>
    <member>
      <name>PortToMonitor</name>
      <value><struct>
        <member>
          <name>IsInput</name>
          <value><boolean>0</boolean></value>
          </member>
        <member>
          <name>Node</name>
          <value><i4>4</i4></value>
          </member>
        <member>
          <name>Port</name>
          <value><i4>55</i4></value>
          </member>
        </struct></value>
      </member>
    </struct></value>
  <value><struct>
    <member>
      <name>ClonePort</name>
      <value><struct>
        <member>
          <name>IsInput</name>
          <value><boolean>0</boolean></value>
          </member>
        <member>
          <name>Node</name>
          <value><i4>2</i4></value>
          </member>
        <member>
          <name>Port</name>
          <value><i4>30</i4></value>
          </member>
        </struct></value>
      </member>
    <member>
      <name>PortToMonitor</name>
      <value><struct>
        <member>
          <name>IsInput</name>
          <value><boolean>0</boolean></value>
          </member>
        <member>
          <name>Node</name>
          <value><i4>4</i4></value>
          </member>
```

```
      <member>
        <name>Port</name>
        <value><i4>48</i4></value>
        </member>
      </struct></value>
    </member>
  </struct></value>
<value><struct>
  <member>
    <name>ClonePort</name>
    <value><struct>
      <member>
        <name>IsInput</name>
        <value><boolean>0</boolean></value>
        </member>
      <member>
        <name>Node</name>
        <value><i4>2</i4></value>
        </member>
      <member>
        <name>Port</name>
        <value><i4>31</i4></value>
        </member>
      </struct></value>
    </member>
  <member>
    <name>PortToMonitor</name>
    <value><struct>
      <member>
        <name>IsInput</name>
        <value><boolean>0</boolean></value>
        </member>
      <member>
        <name>Node</name>
        <value><i4>4</i4></value>
        </member>
      <member>
        <name>Port</name>
        <value><i4>49</i4></value>
        </member>
      </struct></value>
    </member>
  </struct></value>
<value><struct>
  <member>
    <name>ClonePort</name>
    <value><struct>
      <member>
        <name>IsInput</name>
        <value><boolean>0</boolean></value>
        </member>
      <member>
        <name>Node</name>
        <value><i4>2</i4></value>
        </member>
      <member>
        <name>Port</name>
        <value><i4>32</i4></value>
```

```
        </member>
      </struct></value>
    </member>
  <member>
    <name>PortToMonitor</name>
    <value><struct>
      <member>
        <name>IsInput</name>
        <value><boolean>0</boolean></value>
        </member>
      <member>
        <name>Node</name>
        <value><i4>4</i4></value>
        </member>
      <member>
        <name>Port</name>
        <value><i4>50</i4></value>
        </member>
      </struct></value>
    </member>
  </struct></value>
<value><struct>
  <member>
    <name>ClonePort</name>
    <value><struct>
      <member>
        <name>IsInput</name>
        <value><boolean>0</boolean></value>
        </member>
      <member>
        <name>Node</name>
        <value><i4>2</i4></value>
        </member>
      <member>
        <name>Port</name>
        <value><i4>33</i4></value>
        </member>
      </struct></value>
    </member>
  <member>
    <name>PortToMonitor</name>
    <value><struct>
      <member>
        <name>IsInput</name>
        <value><boolean>0</boolean></value>
        </member>
      <member>
        <name>Node</name>
        <value><i4>4</i4></value>
        </member>
      <member>
        <name>Port</name>
        <value><i4>51</i4></value>
        </member>
      </struct></value>
    </member>
  </struct></value>
</data></array></value>
```

```
    </data></array></value>
  </param>
</params>
</methodResponse>
```

As you can see, this is a lot of information, and we need to search through it on receipt of the information and extract the useful bits.

The bellow is the python function used to search through this information using the xml.etree.ElemenTree module and extract only the bits we are interested, namely the presence of clone port identifiers relating to the IFBs for the FB & IFB control interface (30, 34, 44)

Python

```python
from flask import Flask, render_template, request, jsonify
import requests
import xml.etree.ElementTree as ET

def process_output_response(response):
  root = ET.fromstring(response)
  active_ifbs = []

  for value in root.findall('.//value'):
    if value.find('array') is not None:
      data_values = value.find('array').find('data').findall('value')
      for data_value in data_values:
        struct = data_value.find('struct')
        if struct is not None:
          clone_port_node = struct.find(".//member[name='ClonePort']")
          port_to_monitor_node = struct.find(".//member[name='PortToMonitor']")
          if clone_port_node is not None and port_to_monitor_node is not None:
            clone_port_values = clone_port_node.find('value').find('struct')
            if clone_port_values is not None:
              clone_port_i4 = clone_port_values.findall(".//i4")[1].text
              first_i4_value = clone_port_values.findall(".//i4")[0].text

              # Check for IFB activation based on clone_port_i4
              if clone_port_i4 == '30':
                active_ifbs.append('IFB1')
              if clone_port_i4 == '34':
                active_ifbs.append('IFB2')
              if clone_port_i4 == '44':
                active_ifbs.append('IFB3')
            else:
              print("ClonePort struct not found! Using default value for i4")
```

```python
    return active_ifbs
```

The output of this function as seen by the front end of the server is as follow;

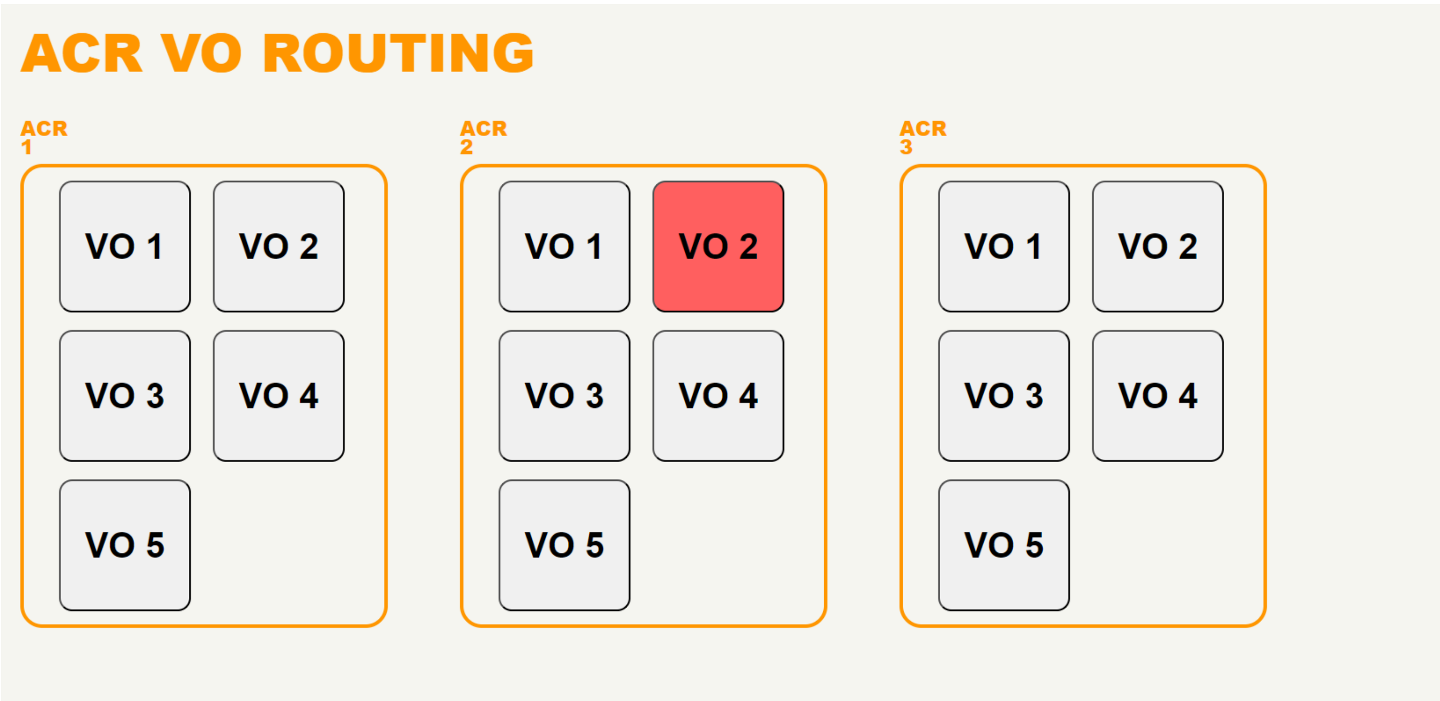
```json
{
  "IFB1": true,
  "IFB2": true,
  "IFB3": false
}
```

This is a much more manageable and useful string of information for the webpage to process, compared to the output provided by the router (as seen above).

2.4 Communication and handling payloads

In this next section, we will follow the process of sending a request to a server, and handling response messages from start to finish, in order of processes to hopefully clarify an example workflow.

Below is the front end for a Flask Server controlling the routing of Voice Over Booths in Park Royal;



The control interface has two functions.

1 - Allows the user the ability to press buttons to route between voice over booths and the relevant ACR
2 - Displays the current routing in the audio router by highlighting the corresponding destination in red.

Above we can see that currently 'VO 2' is routed to 'ACR 2'.

Let's start with a section of the html page;

HTML↴

```
<!DOCTYPE html>
<html>
<head>
    <title>ACR VO Routing</title>
```

```
    <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='VORoutingStyles.css') }}">
        <script>
          function get_crosspoint_statuses() {
                var xhr = new XMLHttpRequest();
                xhr.open('POST', '/get_crosspoint_statuses');
                xhr.send();

                xhr.onload = function() {
                  if (xhr.status === 200) {
                        // Parse the JSON response from the server
                        var response = JSON.parse(xhr.responseText);

                        // Call the function to update the button status
                        updateButtonStatuses(response);
                  } else {
                        alert('Request failed. Status: ' + xhr.status);
                  }
                };
          }

          // Function to update the button status based on the response
          function updateButtonStatuses(response) {
                var statusButton11 = document.getElementById("ACR1_VO1_button");
                var statusButton12 = document.getElementById("ACR1_VO2_button");
                var statusButton13 = document.getElementById("ACR1_VO3_button");
                var statusButton14 = document.getElementById("ACR1_VO4_button");
                var statusButton15 = document.getElementById("ACR1_VO5_button");
                var statusButton21 = document.getElementById("ACR2_VO1_button");
                var statusButton22 = document.getElementById("ACR2_VO2_button");
                var statusButton23 = document.getElementById("ACR2_VO3_button");
                var statusButton24 = document.getElementById("ACR2_VO4_button");
                var statusButton25 = document.getElementById("ACR2_VO5_button");
                var statusButton31 = document.getElementById("ACR3_VO1_button");
                var statusButton32 = document.getElementById("ACR3_VO2_button");
                var statusButton33 = document.getElementById("ACR3_VO3_button");
                var statusButton34 = document.getElementById("ACR3_VO4_button");
                var statusButton35 = document.getElementById("ACR3_VO5_button");

                if (response['ACR1_VO1'] === true) {
                  statusButton11.style.backgroundColor = "#ff6363";
                  statusButton11.innerHTML = "VO 1";
                } else {
                  statusButton11.style.backgroundColor = "#f0f0f0";
                  statusButton11.innerHTML = "VO 1";
                }

# ... Continued for all VO/ACR combinations

          }

          // Start the interval to run the function every 2 seconds (2000 milliseconds)
          setInterval(get_crosspoint_statuses, 2000);
        </script>
</head>
<body>
    <div class="xl-large" id="Page_Name">
        <h2>ACR VO Routing</h2>
    </div>
```

```
    <div class="ACR_1-container">
        <div>
            <h3> ACR 1 </h3>
        </div>
        <div class="ACR_1-box">
                                <button id="ACR1_VO1_button" {% if crosspoint_statuses['ACR1_VO1'] %}class="button-on"{% else
%}class="button-off"{% endif %} onclick="handleButtonClick('ACR1_VO1')">VO 1</button>
                                <button id="ACR1_VO2_button" {% if crosspoint_statuses['ACR1_VO2'] %}class="button-on"{% else
%}class="button-off"{% endif %} onclick="handleButtonClick('ACR1_VO2')">VO 2</button>
                                <button id="ACR1_VO3_button" {% if crosspoint_statuses['ACR1_VO3'] %}class="button-on"{% else
%}class="button-off"{% endif %} onclick="handleButtonClick('ACR1_VO3')">VO 3</button>
                                <button id="ACR1_VO4_button" {% if crosspoint_statuses['ACR1_VO4'] %}class="button-on"{% else
%}class="button-off"{% endif %} onclick="handleButtonClick('ACR1_VO4')">VO 4</button>
                                <button id="ACR1_VO5_button" {% if crosspoint_statuses['ACR1_VO5'] %}class="button-on"{% else
%}class="button-off"{% endif %} onclick="handleButtonClick('ACR1_VO5')">VO 5</button>
        </div>
    </div>

# ... Continued for all VO/ACR Combinations

        </div>
    </div>

    <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
    <script>

          function handleButtonClick(crosspointId) {
                var button = document.getElementById(crosspointId + '_button');
                if (button.classList.contains('button-on')) {
                  // Button is currently on, turn it off
                  turnSDIOff(crosspointId);
                } else {
                  // Button is currently off, turn it on
                  turnSDIOn(crosspointId);
                }
          }

          function turnSDIOn(crosspointId) {
                var xhr = new XMLHttpRequest();
                xhr.open('POST', '/' + crosspointId + '_on');
                xhr.send();

                xhr.onload = function() {
                  if (xhr.status === 200) {
                        // Request was successful, update button state to 'on'
                        document.getElementById(crosspointId + '_button').classList.add('button-on');
                        document.getElementById(crosspointId + '_button').classList.remove('button-off');
                  } else {
                        // Request failed, handle the error if needed
                        alert('Request failed. Status: ' + xhr.status);
                  }
                };
          }

          function turnSDIOff(crosspointId) {
                var xhr = new XMLHttpRequest();
                xhr.open('POST', '/' + crosspointId + '_off');
```

```
        xhr.send();

        xhr.onload = function() {
          if (xhr.status === 200) {
                // Request was successful, update button state to 'off'
                document.getElementById(crosspointId + '_button').classList.add('button-off');
                document.getElementById(crosspointId + '_button').classList.remove('button-on');
          } else {
                // Request failed, handle the error if needed
                alert('Request failed. Status: ' + xhr.status);
          }
        };
  }

  function handleCheckboxChange(checkbox) {
        if (checkbox.checked) {
          if (checkbox.id === "ACR1_VO1_on") {
                ACR1_VO1_on();
          } else if (checkbox.id === "ACR1_VO2_on") {
                ACR1_VO2_on();
          } else if (checkbox.id === "ACR1_VO3_on") {
                ACR1_VO3_on_on();
          } else if (checkbox.id === "ACR1_VO4_on") {
                ACR1_VO4_on_on();
          } else if (checkbox.id === "ACR1_VO5_on") {
                ACR1_VO5_on_on();
          }
        } else {
          if (checkbox.id === "ACR1_VO1_on") {
                ACR1_VO1_off();
          } else if (checkbox.id === "ACR1_VO2_on") {
                ACR1_VO2_off();
          } else if (checkbox.id === "ACR1_VO3_on") {
                ACR1_VO3_off();
          } else if (checkbox.id === "ACR1_VO4_on") {
                ACR1_VO4_off();
          } else if (checkbox.id === "ACR1_VO5_on") {
                ACR1_VO5_off();
          }
        }
  }


  function ACR1_VO1_on() {
        var xhr = new XMLHttpRequest();
        xhr.open('POST', '/ACR1_VO1_on');
        xhr.send();

        xhr.onload = function() {
          if (xhr.status === 200) {
                document.getElementById("response").innerHTML = xhr.responseText;
          } else {
                alert('Request failed. Status: ' + xhr.status);
          }
        };
  }

  function ACR1_VO1_off() {
```

```
            var xhr = new XMLHttpRequest();
            xhr.open('POST', '/ACR1_VO1_off');
            xhr.send();

            xhr.onload = function() {
              if (xhr.status === 200) {
                    document.getElementById("response").innerHTML = xhr.responseText;
              } else {
                    alert('Request failed. Status: ' + xhr.status);
              }
            };
        }

# ... continued for all VO/ACR functions

        function ACR2_VO5_on() {
            var xhr = new XMLHttpRequest();
            xhr.open('POST', '/ACR2_VO5_on');
            xhr.send();

            xhr.onload = function() {
              if (xhr.status === 200) {
                    document.getElementById("response").innerHTML = xhr.responseText;
              } else {
                    alert('Request failed. Status: ' + xhr.status);
              }
            };
        }

        function ACR2_VO5_off() {
            var xhr = new XMLHttpRequest();
            xhr.open('POST', '/ACR2_VO5_off');
            xhr.send();

            xhr.onload = function() {
              if (xhr.status === 200) {
                    document.getElementById("response").innerHTML = xhr.responseText;
              } else {
                    alert('Request failed. Status: ' + xhr.status);
              }
            };
        }

    </script>
</body>
</html>
```

When the page loads, it will run all the script found in the <head> section at user determined intervals. In the above example we have one function;

`get_crosspoint_statuses()`

This will action the endpoint /get_crosspoint_statuses in our python script;

Python⤵

```python
@app.route('/get_crosspoint_statuses', methods=['POST', 'GET'])

def get_crosspoint_statuses():
    script_dir = os.path.dirname(os.path.abspath(__file__))
    server_address = 'http://127.0.0.1:8193'
    headers = {'Content-Type': 'text/xml'}
    # Define the XML files to send for checking crosspoint statuses
    xml_files = {
        'ACR1_VO1_GetXpStatus.xml': 'ACR1_VO1',
        'ACR1_VO2_GetXpStatus.xml': 'ACR1_VO2',
        'ACR1_VO3_GetXpStatus.xml': 'ACR1_VO3',
        'ACR1_VO4_GetXpStatus.xml': 'ACR1_VO4',
        'ACR1_VO5_GetXpStatus.xml': 'ACR1_VO5',
        'ACR2_VO1_GetXpStatus.xml': 'ACR2_VO1',
        'ACR2_VO2_GetXpStatus.xml': 'ACR2_VO2',
        'ACR2_VO3_GetXpStatus.xml': 'ACR2_VO3',
        'ACR2_VO4_GetXpStatus.xml': 'ACR2_VO4',
        'ACR2_VO5_GetXpStatus.xml': 'ACR2_VO5',
    }

    crosspoint_statuses = {}

    # Send a request to each XML file and store the status in the dictionary
    for xml_file, key in xml_files.items():
        xml_path = os.path.join(script_dir, 'xml', xml_file)
        with open(xml_path, 'r') as f:
            xml_data = f.read()

        try:
            response = requests.post(server_address, data=xml_data, headers=headers)
            response.raise_for_status()
            # Get the response text and process it
            response_text = response.text
            has_error, message = process_boolean_response(response_text)
            # Store the status in the crosspoint statuses dictionary
            crosspoint_statuses[key] = not has_error
        except requests.exceptions.RequestException as e:
            # Handle the issues if False
            crosspoint_statuses[key] = False

    return jsonify(crosspoint_statuses)

##############################
# Checking response messages #
##############################
```

```python
def process_boolean_response(response):
    root = ET.fromstring(response)
    boolean_value = None

    for data in root.findall('./params/param/value/array/data'):
        for value in data.findall('value'):
            if value.find('boolean') is not None:
                boolean_value = value.find('boolean').text.strip() == "0"
                break
        if boolean_value is not None:
            break

    if boolean_value is None:
        return (False, "No boolean value found")
    elif boolean_value:
        return (True, "")
    else:
        return (False, "")
```

The above Python function sends the xml files specified to the router, and relates the response to a dictionary;

```
        'ACR1_VO1_GetXpStatus.xml': 'ACR1_VO1',
```

The responses are then processed by the function 'process_boolean_response(response)'. This function checks the response for wither a 1 or 0 in the i4 section of the xml response string from the server, and appends the result 'True' for '1' and 'False' for '0'.

This information is constructed in JSON format to be accessible by the front end webpage.

The html file gathers this information using the get_crosspoint_statuses function and displays the information extracted from the JSON file as follows;


```
  "ACR1_VO1": false,
  "ACR1_VO2": false,
  "ACR1_VO3": false,
  "ACR1_VO4": false,
  "ACR1_VO5": false,
  "ACR2_VO1": false,
  "ACR2_VO2": true,
  "ACR2_VO3": false,
  "ACR2_VO4": false,
  "ACR2_VO5": false
```

```
}
```

Using this information the get_crosspoint_statuses function sets the color of the buttons.

When the user clicks on the VO1 button in the ACR1 section of the control page, for example;

```
<button id="ACR1_VO1_button" {% if crosspoint_statuses['ACR1_VO1'] %}class="button-on"{% else %}class="button-off"{% endif %} onclick="handleButtonClick('ACR1_VO1')">VO 1</button>
```

the following JavaScript function will be executed, depending on the current status;

```
        function ACR1_VO1_on() {
             var xhr = new XMLHttpRequest();
             xhr.open('POST', '/ACR1_VO1_on');
             xhr.send();

             xhr.onload = function() {
               if (xhr.status === 200) {
                    document.getElementById("response").innerHTML = xhr.responseText;
               } else {
                    alert('Request failed. Status: ' + xhr.status);
               }
             };
        }

        function ACR1_VO1_off() {
             var xhr = new XMLHttpRequest();
             xhr.open('POST', '/ACR1_VO1_off');
             xhr.send();

             xhr.onload = function() {
               if (xhr.status === 200) {
                    document.getElementById("response").innerHTML = xhr.responseText;
               } else {
                    alert('Request failed. Status: ' + xhr.status);
               }
             };
        }
```

This requests the function /ACR1_VO1_on (or off) from the Flask server script;

```python
@app.route('/ACR1_VO1_on', methods=['POST', 'GET'])
def ACR1_VO1_on():
    script_dir = os.path.dirname(os.path.abspath(__file__))
    xml_files = ['ACR1_to_VO1_SetXP.xml', 'VO1_to_ACR1_SetXP.xml']
    server_address = 'http://127.0.0.1:8193'
    headers = {'Content-Type': 'text/xml'}
    for xml_file in xml_files:
        xml_file_path = os.path.join(script_dir, 'xml', xml_file)
        with open(xml_file_path, 'r') as f:
            xml_data = f.read()
        requests.post(server_address, data=xml_data, headers=headers)
    return ' '
```

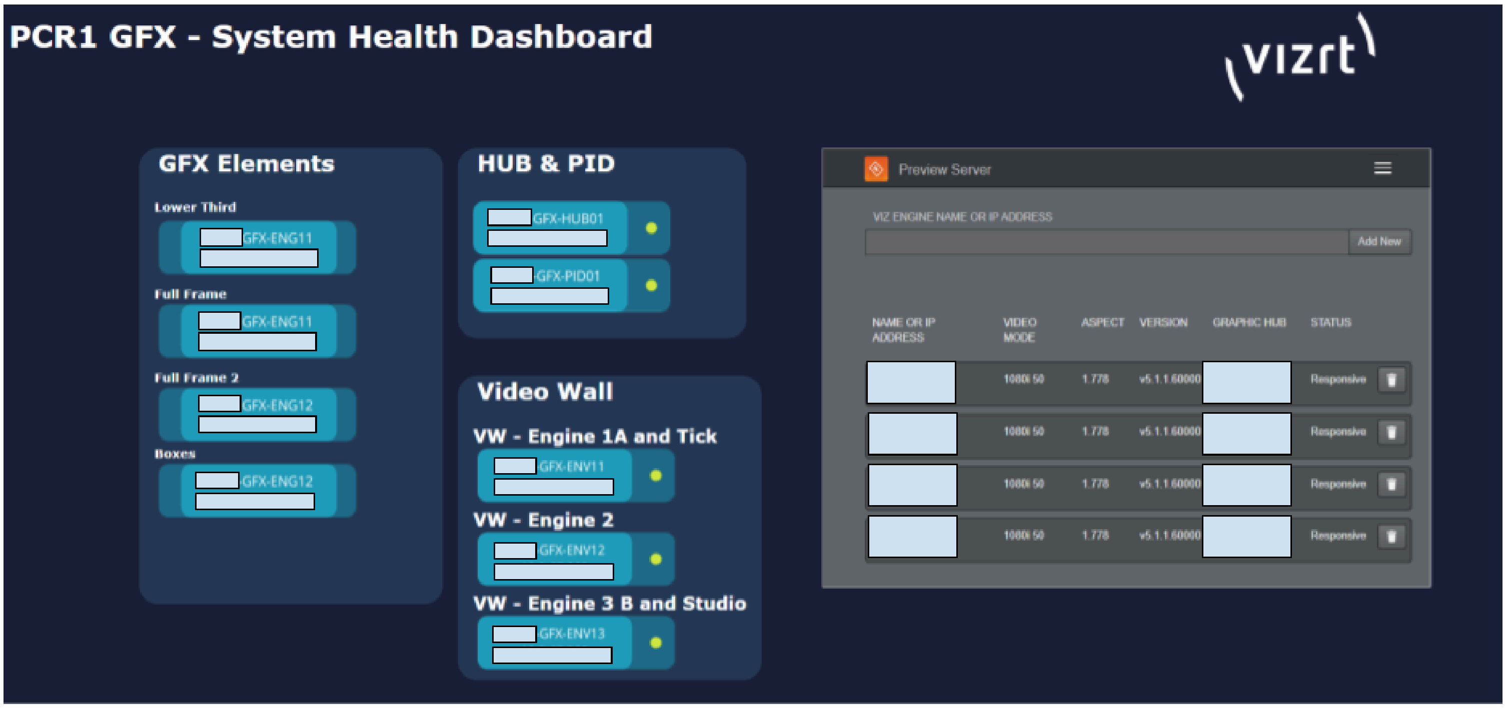
The 'get_crosspoint_statuses' command is run every 2 seconds in this script, so if the route was successfully made, the function will pick this up in the next JSON response file and update the buttons.

	Remember it is important to have your router change actions separate from your status update functions. They need to be acting independently for two reasons; 1 - Reduce false positives, if the button color changes based off of the button press action, you would only be confirming you pressed the button. 2 - If the crosspoint, or any action, was made from somewhere else/ someone else, you need your interface to be updated.

2.5 Providing User Feedback

User feedback as we've discussed above is imperative to control systems. IF we don't have a clear picture of the current state of a system, we can't even start to control it. Below we will highlight some methods of gathering and providing user feedback.

The first example we will look at is a simple monitoring page.



The above example is an interface that displays the current state of several hardware servers and web servers running services on a the graphics network.

Bellow is the server script in full;

Python

```python
from flask import Flask, render_template, request, jsonify
import subprocess
import paramiko

app = Flask(__name__)

@app.route('/')
def home():
    return render_template('index.html')
```

```python
vidwall_servers = [
    {'id': 'GFX-ENV11', 'ip': 'xx.xx.xx.xx', 'port': 0000},
    {'id': 'GFX-ENV12', 'ip': 'xx.xx.xx.xx', 'port': 0000},
    {'id': 'GFX-ENV13', 'ip': 'xx.xx.xx.xx', 'port': 0000}
]

hubpid_servers = [
    {'id': 'GFX-HUB01', 'ip': 'xx.xx.xx.xx', 'port': 0000},
    {'id': 'GFX-PID01', 'ip': 'xx.xx.xx.xx', 'port': 0000}
]

gfxg1_servers = [
    {'id': 'GFX-ENG116100', 'ip': 'xx.xx.xx.xx', 'port': 6100},
    {'id': 'GFX-ENG116800', 'ip': 'xx.xx.xx.xx', 'port': 6800},
    {'id': 'GFX-ENG126100', 'ip': 'xx.xx.xx.xx', 'port': 6100},
    {'id': 'GFX-ENG126800', 'ip': 'xx.xx.xx.xx', 'port': 6800}
]

def check_server(server):
    command = f'ping {server["ip"]} -n 2'
    result = subprocess.run(['powershell', '-Command', command], capture_output=True)
    output = result.stdout.decode('utf-8')
    return 'Reply from' in output

@app.route('/check_vidwall_servers', methods=['POST'])
def check_vidwall_servers():
    try:
        vidwall_results = []

        for server in vidwall_servers:
            is_server_up = check_server(server)
            vidwall_results.append({'serverId': server['id'], 'status': is_server_up})

        return jsonify({'vidwall': vidwall_results,})
    except Exception as e:
        return jsonify(error=str(e)), 500

@app.route('/check_hubpid_servers', methods=['POST'])
def check_hubpid_servers():
    try:
        hubpid_results = []

        for server in hubpid_servers:
            is_server_up = check_server(server)
            hubpid_results.append({'serverId': server['id'], 'status': is_server_up})

        return jsonify({'hubpid': hubpid_results,})
```

```
    except Exception as e:
        return jsonify(error=str(e)), 500

@app.route('/check_gfxg1_servers', methods=['POST'])
def check_gfxg1_servers():
    try:
        gfxg1_results = []

        for server in gfxg1_servers:
            command = f'Test-NetConnection -ComputerName {server["ip"]} -Port {server["port"]}'
            result = subprocess.run(['powershell', '-Command', command], capture_output=True)
            output = result.stdout.decode('utf-8')
            is_server_up = 'TcpTestSucceeded : True' in output
            gfxg1_results.append({'serverId': server['id'], 'status': is_server_up})

        return jsonify({'gfxg1': gfxg1_results})
    except Exception as e:
        return jsonify(error=str(e)), 500

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=5072)
```

The html page is quite basic, just changing buttons from green, to red, to neutral based on the response from the server, so we won't go into it in much detail.

In the above python script, we use the subprocess module to start processes that can run a line of code and retrieve the output for use in the script.

Using the above example '/check_gfxg1_servers' we use the subprocess module to construct a command and run it in Windows Powershell.
The constructed command will look as follows for the Lower Third web server running on [redacted];

`Test-NetConnection -ComputerName xx.xx.xx.xx -Port 6100'`

The response from Powershell will look like this;

```
ComputerName     : xx.xx.xx.xx
RemoteAddress    : xx.xx.xx.xx
RemotePort       : 6100
InterfaceAlias   : WiFi
SourceAddress    : xx.xx.xx.xx
TcpTestSucceeded : True
```

We examine the output in our python function to extract the TcpTestSuccessed value;

```
is_server_up = 'TcpTestSucceeded : True' in output
```

We then convert this information, along with all other addresses into a JSON file to make it easily understandable for out front end;

```
return jsonify({'gfxg1': gfxg1_results})
```

The webpage received this information string;

```
0: {serverId: "GFX-ENG116100", status: true}
1: {serverId: "GFX-ENG116800", status: true}
2: {serverId: "GFX-ENG126100", status: true}
3: {serverId: "GFX-ENG126800", status: true}
```

This information is easily understandable, and can be used to update button statuses.

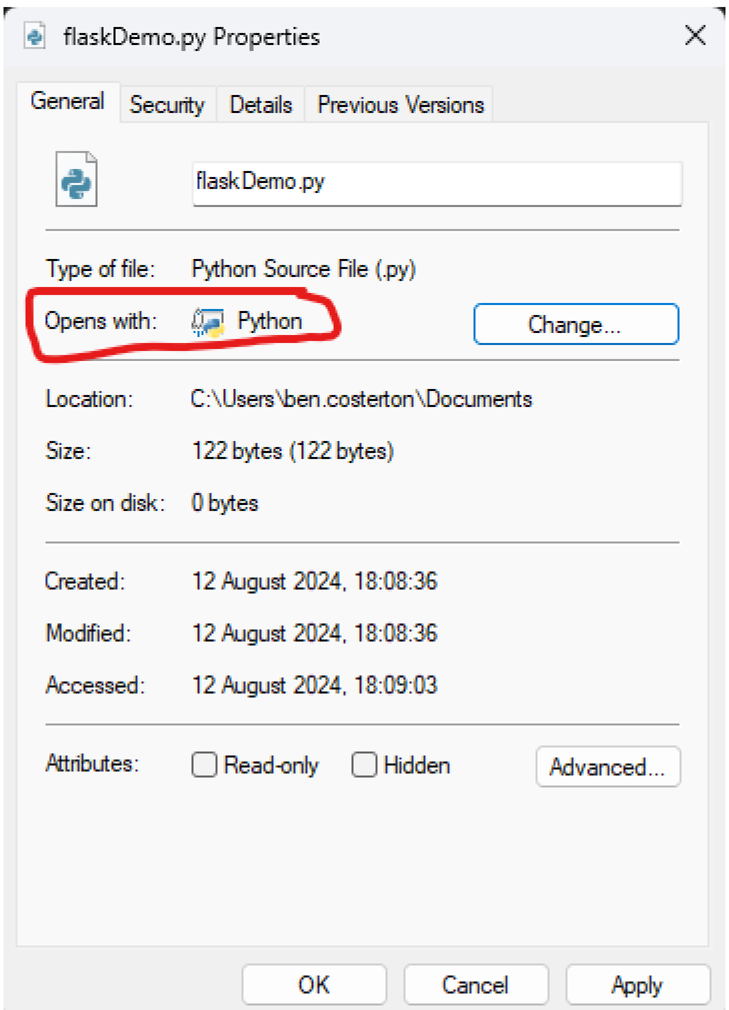
2.6 Managing Web Servers

It is recommended to at least make sure that the Python files are set up in an executable fashion.

As stated earlier in the installation instructions for Flask, once Python is installed on the local system, you should be able to run the script with the Python Launcher.

Windows Setup

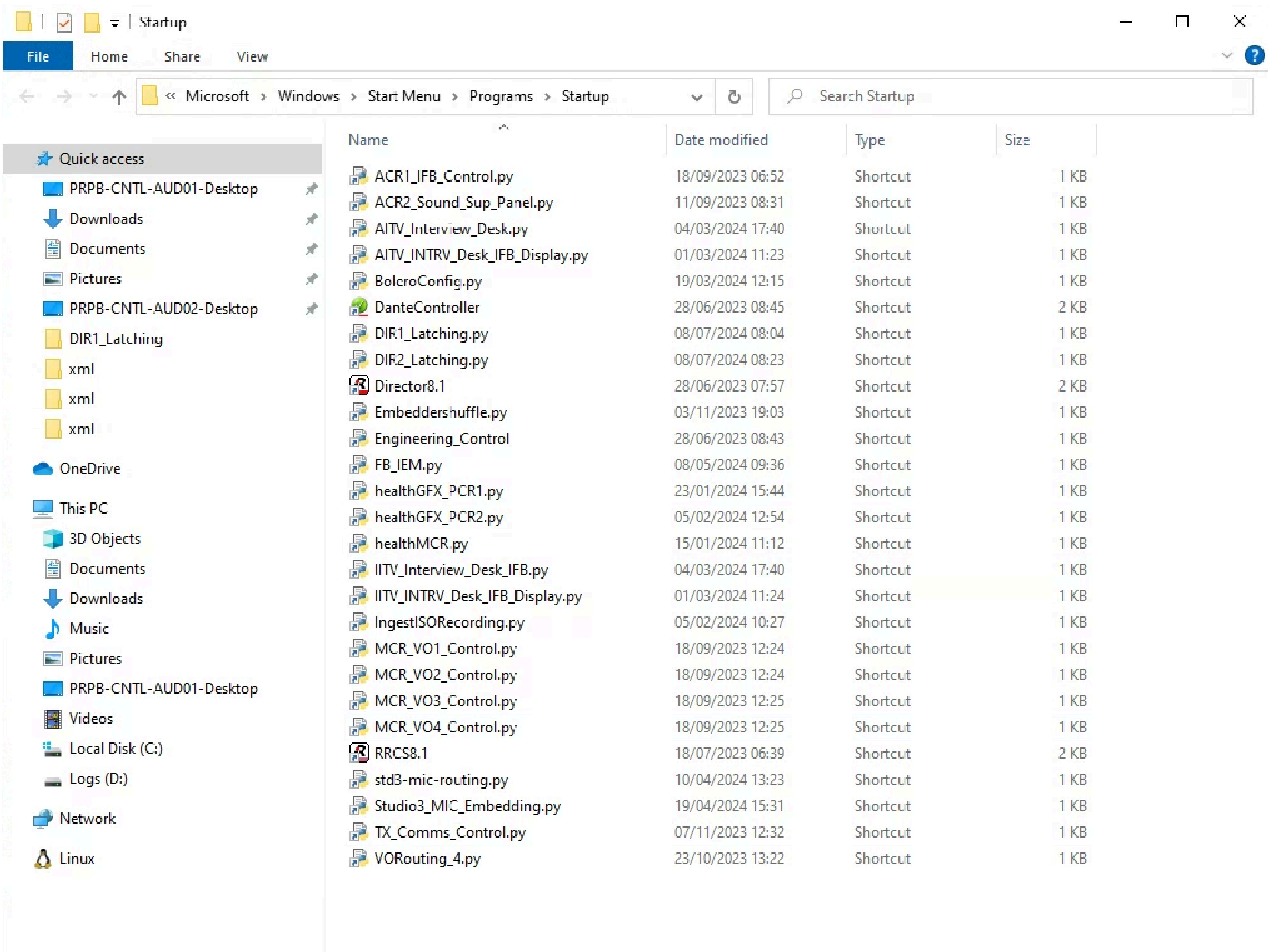
For Windows servers running Flask script, the following setup is recommended;



Set the default application for .py files to 'Python'.
This will allow the script to be executed instead of opening with a document editor.

Having .py files default to Python Launcher will allow you to add python scripts to startup operations:

'RUN - shell:startup' will open the startup folder in Windows;



.py field found in the ***Microsoft>Windows>Start Menu>Programs>Startup*** folder will be run on system startup. This is the safest way to secure web server up time in the event of unplanned system reboots.

3.0 Interfaces

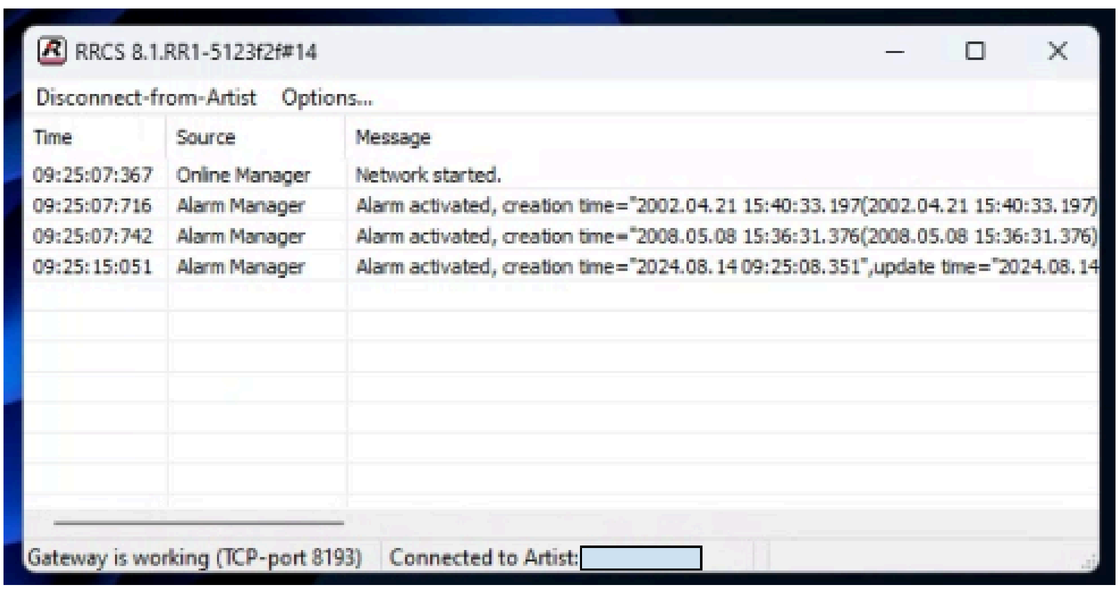
3.1 Interface overview - A definition

When the user guide refers to an 'Interface' it refers to any piece of software or service sitting between the control system and the end point destination.

Most manufacturers of routers for example will provide an out-of-the box solution that can be used to build a control system on top of. This can be a piece of software designed to be used directly by the user, software listening for commands on a specified port, or an boilerplate API for example.

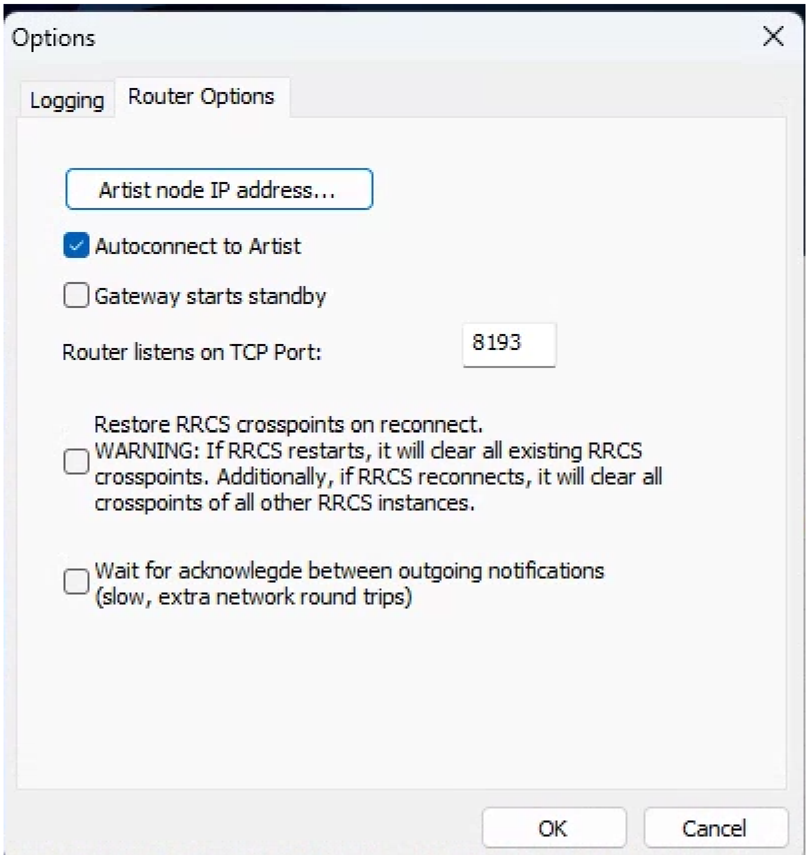
3.2 Riedel RRCS

Riedel's RRCS is how we connect any control system up to the Riedel Artist audio matrix. It is a Windows application that needs to run on a PC with network visibility of one of the Artist Node CUP IP addresses.

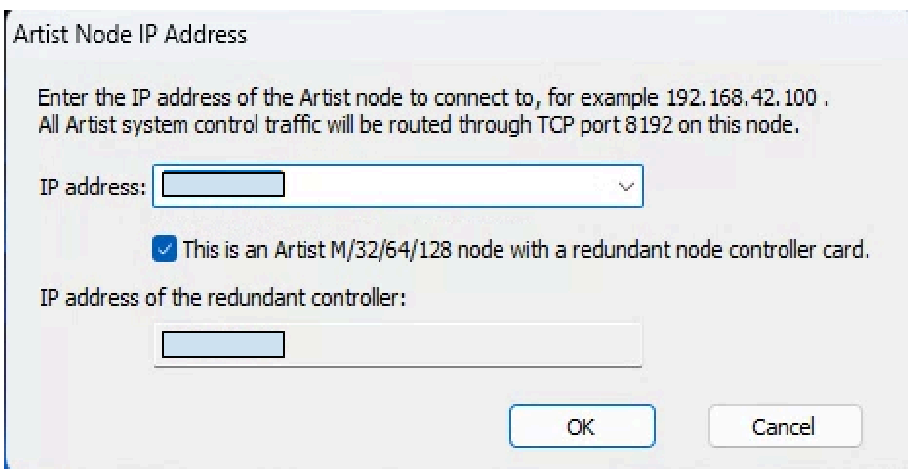


The application can be downloaded from the Riedel Download Center at MyRiedel.net.

Upon opening the application for the first time, you will need to connect it to the Artist frame.
Open the 'Options...' tap from the menu bar, you will see this window;

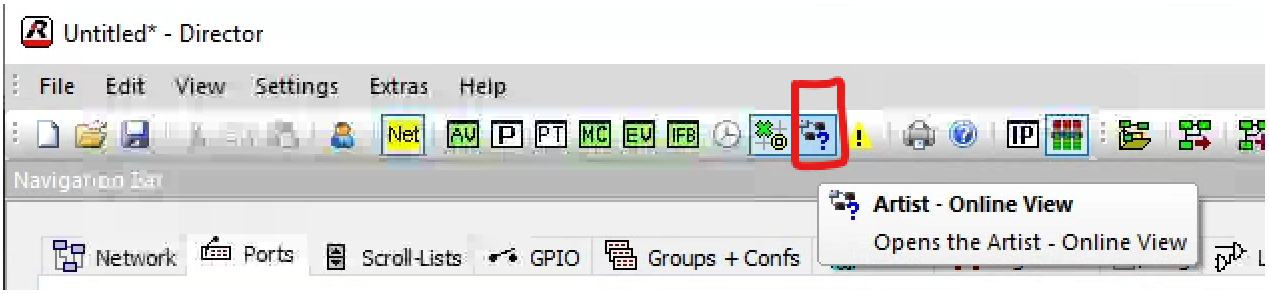


Click on 'Artist node IP address...



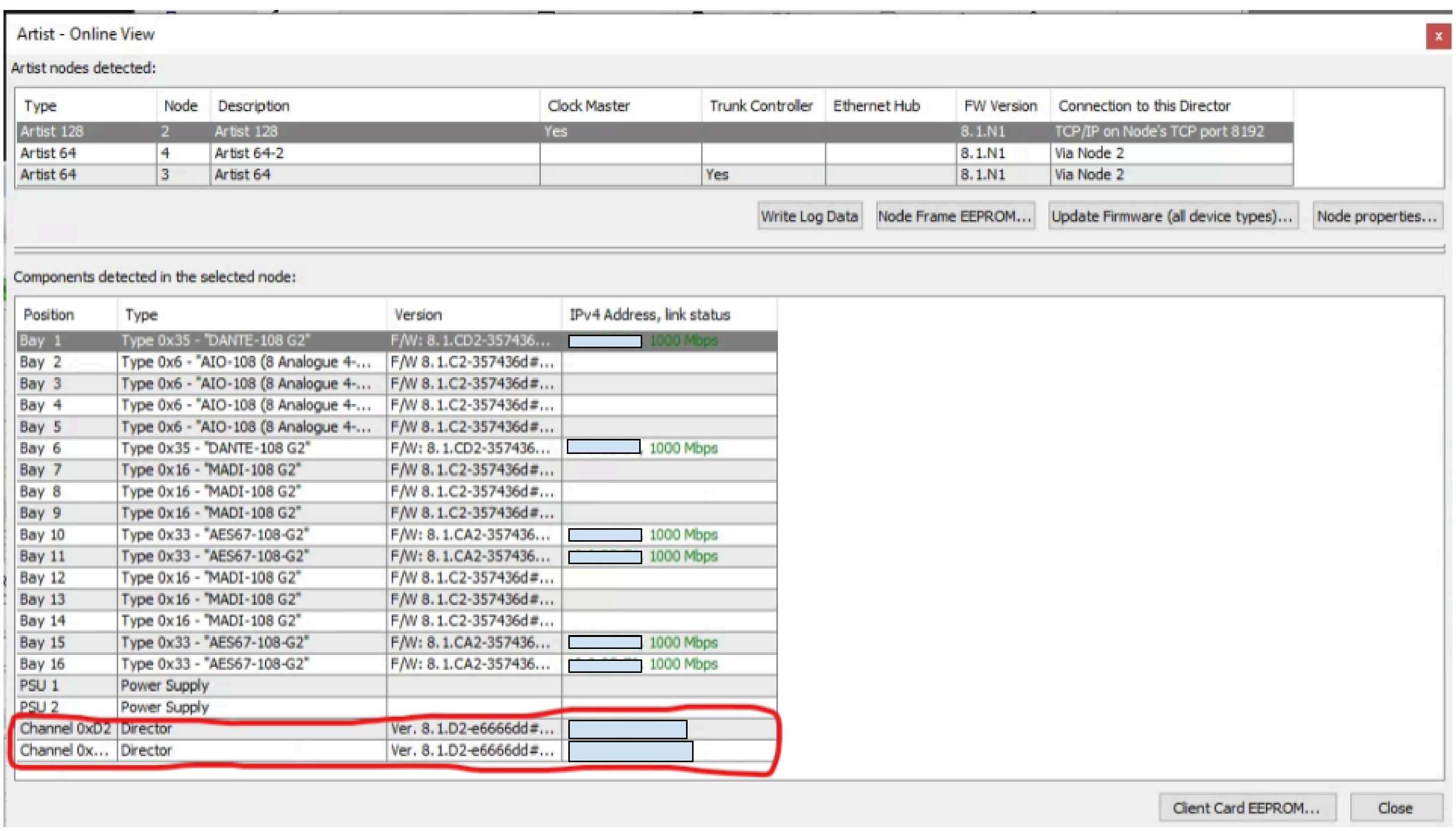
From within the 'Artist Node IP Address' window, enter the IP address of one of the nodes in the Artist system. It does not matter which IP address you use from the frame setup.

	Be aware, you can only connect 4 applications to each Artist Node. Applications include Director, Trunk Navigator, and RRCS. Check your Node Config from the 'Online View' in the Riedel Director software.



By opening 'Online View' and entering password if required, you will see the Riedel Director Online View:

From 'Online View' we can see which applications are connected to which Node in the system.

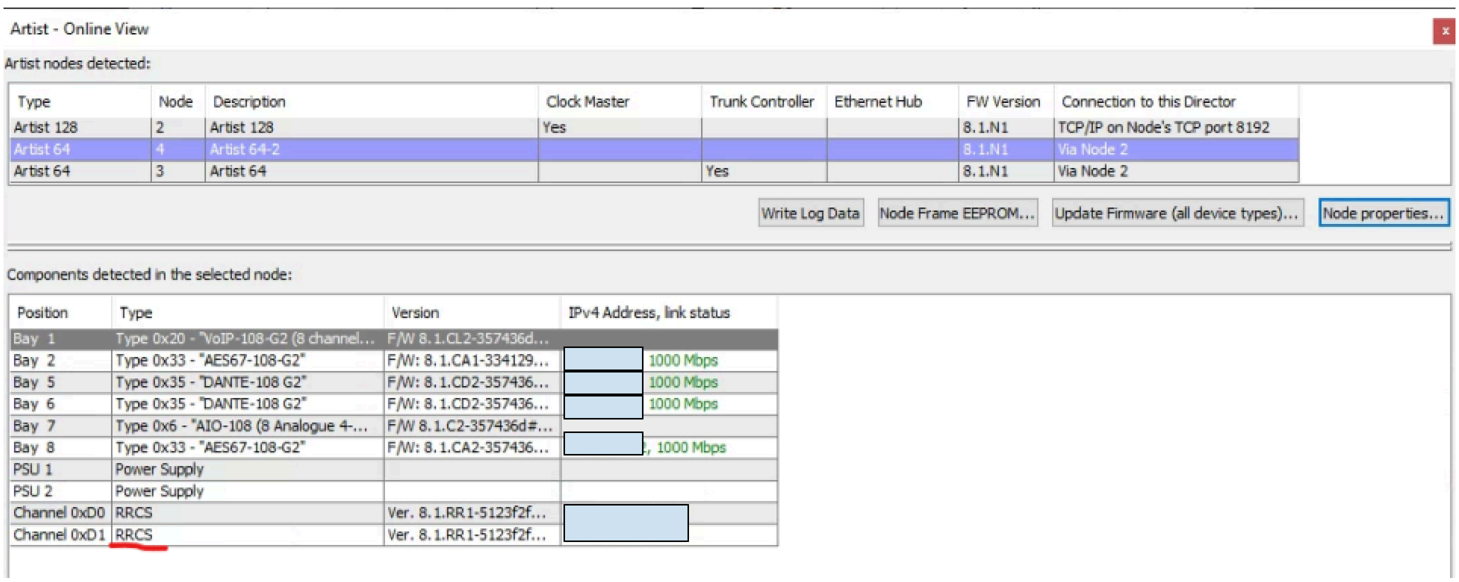


In the above example, two computers are connected to the Artist 'Node 2' with the Riedel Director software. As each Node can be connected to 4 applications simultaneously, we can add an RRCS connection here.

Note the IP address of the Node you wish to connect RRCS to, and enter this information into the RRCS application as seen above. Select Redundant network if you Artist Node is setup with redundant CPU cards.

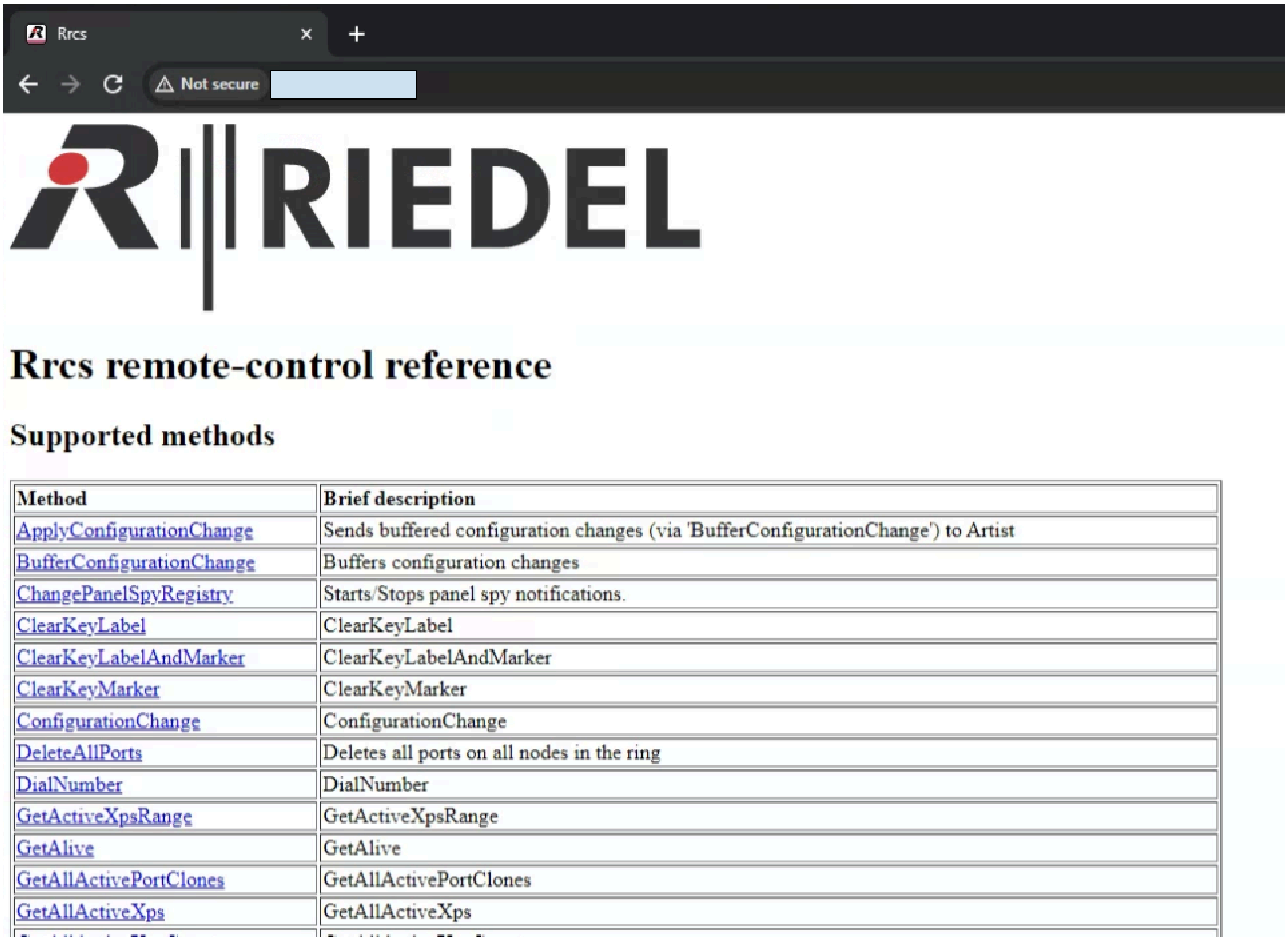
	The 'Router Listen TCP Port' is set to 8193 by default, and should be left this way unless you have a specific reason to change it.

Click 'OK' to close the 'Options' window, and click 'Connect-to-Artist'.
Once complete, you should see the RRCS connection in the 'Online View' from the Riedel Director application under the Node you chose in the options window of RRCS setup;



RRCS is now ready to receive requests at the IP address of the computer hosting the software at the port chosen in the Options window (default 8193).

The software will host a web server of it's own, found at the ip address of the host machine with the port chosen in the steps above;



Rrcs remote-control reference

Supported methods

Method	Brief description
ApplyConfigurationChange	Sends buffered configuration changes (via 'BufferConfigurationChange') to Artist
BufferConfigurationChange	Buffers configuration changes
ChangePanelSpyRegistry	Starts/Stops panel spy notifications.
ClearKeyLabel	ClearKeyLabel
ClearKeyLabelAndMarker	ClearKeyLabelAndMarker
ClearKeyMarker	ClearKeyMarker
ConfigurationChange	ConfigurationChange
DeleteAllPorts	Deletes all ports on all nodes in the ring
DialNumber	DialNumber
GetActiveXpsRange	GetActiveXpsRange
GetAlive	GetAlive
GetAllActivePortClones	GetAllActivePortClones
GetAllActiveXps	GetAllActiveXps

This web server acts as a way to access sample layouts for different types of commands, but can also be used to submit commands to the system manually:

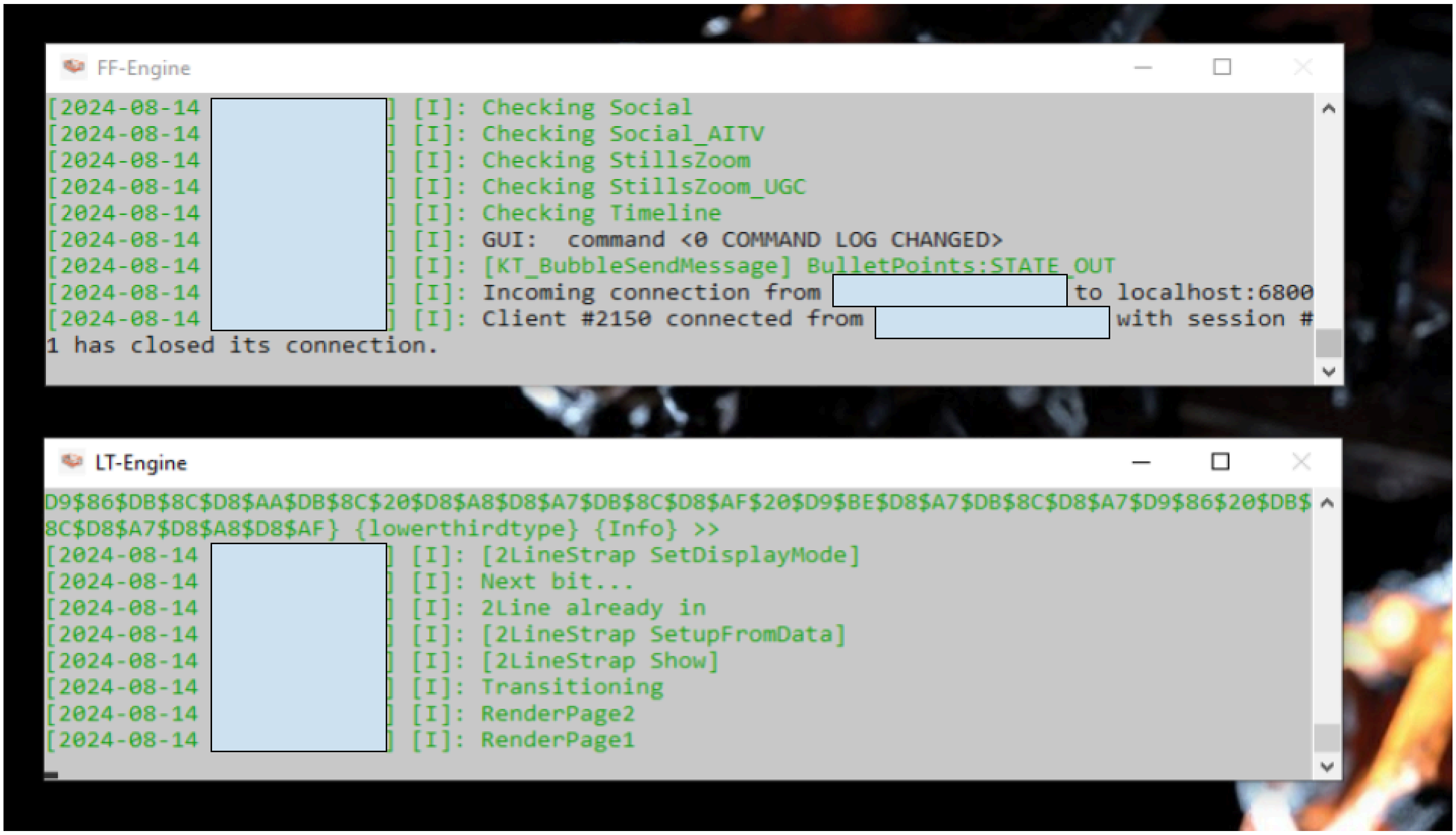
By clicking on one the the 'Methods' you will get access to a sample xml script, and have the option to 'execute' the command to see the output:

From here, it is up to you as the user to choose how you will send commands to RRCS. You can either use a piece of control software that supports Riedel RRCS out of the box, Cerebrum for example, or write an application that can send xml payloads via HTTP request to the address:port you set up above.

This is a good way to test commands in raw text form.

3.3 Viz Web servers

Viz servers used for PCR gallery elements provide this content via web servers running on the physical servers. When logged into graphics engines, you can see the servers running;



These servers run on specific ports on each machine, in the above example, we are looking at the Full-Frame and Lower-Thrird servers running on [redacted]. In this example, the Full-Frame engine is running on port 6100, and the Lower-Third server is running on port 6800.

These addresses are used by Viz Director to obtain the graphics elements needed for a specific show.

We can use these addresses to obtain some basic information outside of the Viz Workflow, for example tracking uptime of the specific servers running on these machines.

4.4 Audinate Dante & Net Audio Controller

Audinates Dante software is not very control system friendly. What I mean by this is Audinate wishes you to use the Dante Controller to control all actpets of the Dante work, however there are workarounds.

Control systems like Cerebrum 'support' Dante out of the box. In practice this guide does not support the use of cerebrum for controlling a Dante network. The implementation is limited, and not very stable.
The main drawbacks are the requirement for the Dante network to remain static. What I mean by this is that once Cerebrum is connected to the Dante network, no device can be added or removed, doing so will change the order of devices from Cerebrums view, and you run the risk of making incorrect crosspoint allocations, and it's not worth the risk.
Furthermore, changes to the Dante system, for example renaming channels will not reflect in Cerebrum until the network is refreshed, which is an annoying task for a developer to do constantly.

There is a tool developed by Chris Risen, 'network-audio-controller' which this guide recommends over Cerebrum.

Found here - https://github.com/chris-ritsen/network-audio-controller, Network Audio Controller is a python application that uses it's own syntax to control the Dante network, importantly using plain text channel names.

Install Net Audio Controller using pythons pip repository;

```
pip install netaudio
```

Windows users will have to remove a module from the netaudio package to get this application to talk to Dante correctly. Follow this guide.

1 - On completion of installation, navigate the following folder on your Windows machine (C:\Users**USER-NAME**\AppData\Local\Programs\Python**PYTHON-VERSION**\Lib\site-packages\netaudio\console)

2 - Open the application.py file in a text editor

3 - Comment out the import of the signal module (line 1)

4 - Comment out the use of the signal module (line 11)

5 - Save the file

Your application.py file should look something like this once edits are complete;

```
#from signal import signal, SIGPIPE, SIG_DFL
from cleo.application import Application

from netaudio.console.commands import (
    ChannelCommand,
    ConfigCommand,
    DeviceCommand,
    SubscriptionCommand,
)

#signal(SIGPIPE, SIG_DFL)


def main() -> int:
    application = Application("netaudio", "0.0.10", complete=True)
    application.add(ChannelCommand())
    application.add(ConfigCommand())
    application.add(DeviceCommand())
    application.add(SubscriptionCommand())

    return application.run()


if __name__ == "__main__":
    main()
```

A list of Netaudio Controller commands can be found on Chris Risen github page (https://github.com/chris-ritsen/network-audio-controller/wiki/Examples).

Use these commands to control the Dante network, and write these into whatever control system you choose.

For example, here is a basic python script using the os module to execute Dante control commands as part of a Flask web server;

Python

```python
from flask import Flask, render_template, jsonify
import subprocess
import json
```

```python
import logging
import os

app = Flask(__name__, static_folder='static')

@app.route('/', methods=['GET', 'POST'])
def index():
    return render_template('std3microuting.html')


#Make routes in Dante
def execute_command(command):
    try:
        output = subprocess.check_output(command, shell=True, text=True)
        logging.debug(f"Command executed successfully: {command}")
        return output
    except subprocess.CalledProcessError as e:
        logging.error(f"Error executing command: {command}. Error: {e}")
        return str(e)

@app.route('/in1->out7', methods=['POST'])
def in1out7():
    command = 'netaudio subscription add --tx-device-name=A32 --tx-channel-name=STD3Wallbox1-11
--rx-channel-name=17_STD3_Wallbox7 --rx-device-name=A32'
    return execute_command(command)

@app.route('/in2->out7', methods=['POST'])
def in2out7():
    command = 'netaudio subscription add --tx-device-name=A32 --tx-channel-name=STD3Wallbox2-12
--rx-channel-name=17_STD3_Wallbox7 --rx-device-name=A32'
    return execute_command(command)

# Remove routes in Dante

@app.route('/remove->out7', methods=['POST'])
def removeout7():
    command = 'netaudio subscription remove --rx-channel-name=17_STD3_Wallbox7
--rx-device-name=A32'
    return execute_command(command)

@app.route('/remove->out8', methods=['POST'])
def removeout8():
    command = 'netaudio subscription remove --rx-channel-name=18_STD3_Wallbox8
--rx-device-name=A32'
    return execute_command(command)
```

```python
# Get sublist from Dante
@app.route('/SubList', methods=['POST', 'GET'])
def get_sub_list():
    commands = [
        'netaudio subscription list'
    ]

    output_data = {}

    for command in commands:
        output = subprocess.check_output(command, shell=True, text=True)
        output_data[command] = output

    # Store the output data in a JSON file
    with open('output.json', 'w') as f:
        json.dump(output_data, f)

    return jsonify(output_data)


if __name__ == '__main__':
    app.run(debug=True, host="0.0.0.0", port=5021)
```

November 2024 update;

I've written a new version of this module here ->
https://github.com/Bencosterton/dante-network-control

Download the .whl file from dist/ (https://github.com/Bencosterton/dante-network-control/tree/master/dist)
and install via pip;

```
python -m pip install netaudio-0.2.0-py3-none-any.whl
```

Changelog (https://github.com/Bencosterton/dante-network-control/blob/master/CHANGELOG.md);
0.0.2 (2024-11-18)
Ben Costerton Update

- Added the device handling for 'subscription list' back in
- Removed a fair bit of mDNS output to clean up terminal
- Removed the connection status appended to the end of all subscription messages

```
- Added logging_config.py to __main__ to reduce the amount of cli output from netaudio
- Cleaned up the SIGPIPE handling so it runs smoother on Windows systems out the box
```

3.5 Operating System Commands

The os module is also great for obtaining information about servers for monitoring purposes. As discussed in examples from section "2.5 Providing user feedback" under the Flask tutorials, we can use system commands to obtain information uptime of servers, and provide this to the user, as well of any other command that can be executed on an OS level.

!	If writing your own control script, be wary of where your script will run. You don't want to write an application using Windows based os commands to find the target server is running on linux!

A suggestion if unsure of target os would be to write os commands twice, once for a Windows target machine, and once for Unix target machine.

Here are some examples of how to import these system commands in multiple languages;

Python⤵

```python
try:
    import win_commands
except ImportError:
    import unix_commands
```

Node.js⤵

```javascript
let commands;

try {
    commands = require('./win_commands');
} catch (error) {
    commands = require('./unix_commands');
}

commands.someFunction();
```

Java⤵

```java
public class Main {
    public static void main(String[] args) {
        Object commands = null;

        try {
```

```
            Class<?> winCommandsClass = Class.forName("WinCommands");
            commands = winCommandsClass.getDeclaredConstructor().newInstance();
        } catch (ClassNotFoundException e) {
            try {
                Class<?> unixCommandsClass = Class.forName("UnixCommands");
                commands = unixCommandsClass.getDeclaredConstructor().newInstance();
            } catch (ClassNotFoundException | InstantiationException | IllegalAccessException |
NoSuchMethodException | InvocationTargetException ex) {
                ex.printStackTrace();
            }
        }

        if (commands != null) {
            try {
                commands.getClass().getMethod("someFunction").invoke(commands);
            } catch (NoSuchMethodException | IllegalAccessException | InvocationTargetException
e) {
                e.printStackTrace();
            }
        } else {
            System.out.println("No suitable commands class found.");
        }
    }
}
```

This way you can deploy a control server using os commands onto any machine, and be sure it will be able to use the correct system based syntax.

3.6 Imagine IP3 LRC

The Imagine IP3 is the main video router in [redacted]. It performs most video routing tasks within the broadcast environment.

Imagine IP3 used LRC (local remote command) as a command protocol.

Imagine IP3 has its own syntax in use that needs to be used when sending commands over LRC.

- **Crosspoint Take (XPOINT)** - a crosspoint command from a source to a destination
  // ~XPOINT:S${SKYPE-1-1};D${TS-MON-1}\
- **Crosspoint Disconnect (XDISCONNECT)** - disconnect an existing crosspoint
  // ~XDISCONNECT:D${TS-MON-1}\
- **Crosspoint Preset (XPRESET)** - preset a crosspoint command
  // ~XPRESET:D#{2};S#{1};U#{1}\
- **Salvo Execution (XSALVO)** - execute a crosspoint salvo
  // ~XSALVO:F${FLAG,FLAG,FLAG}
- **Destination Lock (LOCK)** - Lock or unlock a destination
  // ~LOCK:D${TS-MON-1};V${ON};U#{20}\
- **Destination Protect (PROTECT)** - Protect or unprotect a destination
  // ~PROTECT:D${TS-MON-1};V${ON};U#{20}\
- **Protocol Query (PROTOCOL)** - query the protocol version information
  // ~PROTOCOL?Q${NAME}\
- **Source Query (DEST)**- Query the router for the logical source of a destination
  //~XPOINT?D${TS-MON-1}\

Crosspoint Take (XPOINT)

In it's most basic form, this is how to send a route command to the IP3 router to route the source 'G1-PGM' to the destination 'TS-MON-1';

```
echo -n '~XPOINT:S${G1-PGM};D${TS-MON-1}\n' | nc <IP-ADDRESS> 52116
```

Crosspoint Disconnect (XDISCONNECT)

Bellow is the method of disconnecting a destination from any source. This will leave the source in (Disconnected) state and will display black.

```
echo -n '~XDISCONNECT:D${TS-MON-1}\n' | nc <IP-ADDRESS> 52116
```

Locking and unlocking a destination (LOCK)

Bellow is how to lock and unlock a destination, in this example 'TS-MON-1';

```
#lock destination TS-MON-1
echo -n '~LOCK:D${TS-MON-1};V${ON};U#{20}\n' | nc <IP-ADDRESS> 52116

#unlock destination TS-MON-1
echo -n '~LOCK:D${TS-MON-1};V${OFF};U#{20}\n' | nc <IP-ADDRESS> 52116
```

Protect and unprotect a destination (PROTECT)

Note: Protection can only be removed by the user who initiated the protect status, and therefore is not a recommended function, see 'LOCK' instead

```
#Protect destination TS-MON-1
echo -n '~PROTECT:D${TS-MON-1};V${ON};U#{20}\n' | nc <IP-ADDRESS> 52116

#Unprotect destination TS-MON-1
echo -n '~PROTECT:D${TS-MON-1};V${OFF};U#{20}\n' | nc <IP-ADDRESS> 52116
```

Source Query (XPOINT)

Below command will query which source is currently routed to a destination.

```
echo -n '~XPOINT?D${TS-MON-1}\n' | nc <IP-ADDRESS> 52116
```

Working on Windows

If working on Windows, I would recommend installing WSL (Windows Subsystem for Linux) and running the command with 'wsl -e'. For example, routing 'G1-PGM' to 'TS-MON-1' on Windows with WSL installed would look like this;

```
wsl -e echo -n '~XPOINT:S${G1-PGM};D${TS-MON-1}\n' | nc <IP-ADDRESS> 52116
```

If you are looking to embed the command into a script, say to be executed by the os module in python and you require variable input, this is the syntax solution I found;

```
f'wsl -e bash -c "echo -n \'~XPOINT:S${{{src}}};D${{{dst}}}\\n\' | nc -w 1 <IP-ADDRESS> 52116"'
```

Doing so will open a WSL shell, run the command, and close the shell.

3.7 Lawo Ember+

Lawo Ember+ can be used to control many devices, below we will use examples for Lawo mc² and VPro.

Lawo uses two protocols to control mc² devices;
- RemoteMNOPL
- Ember+

RemoteMNOPL;

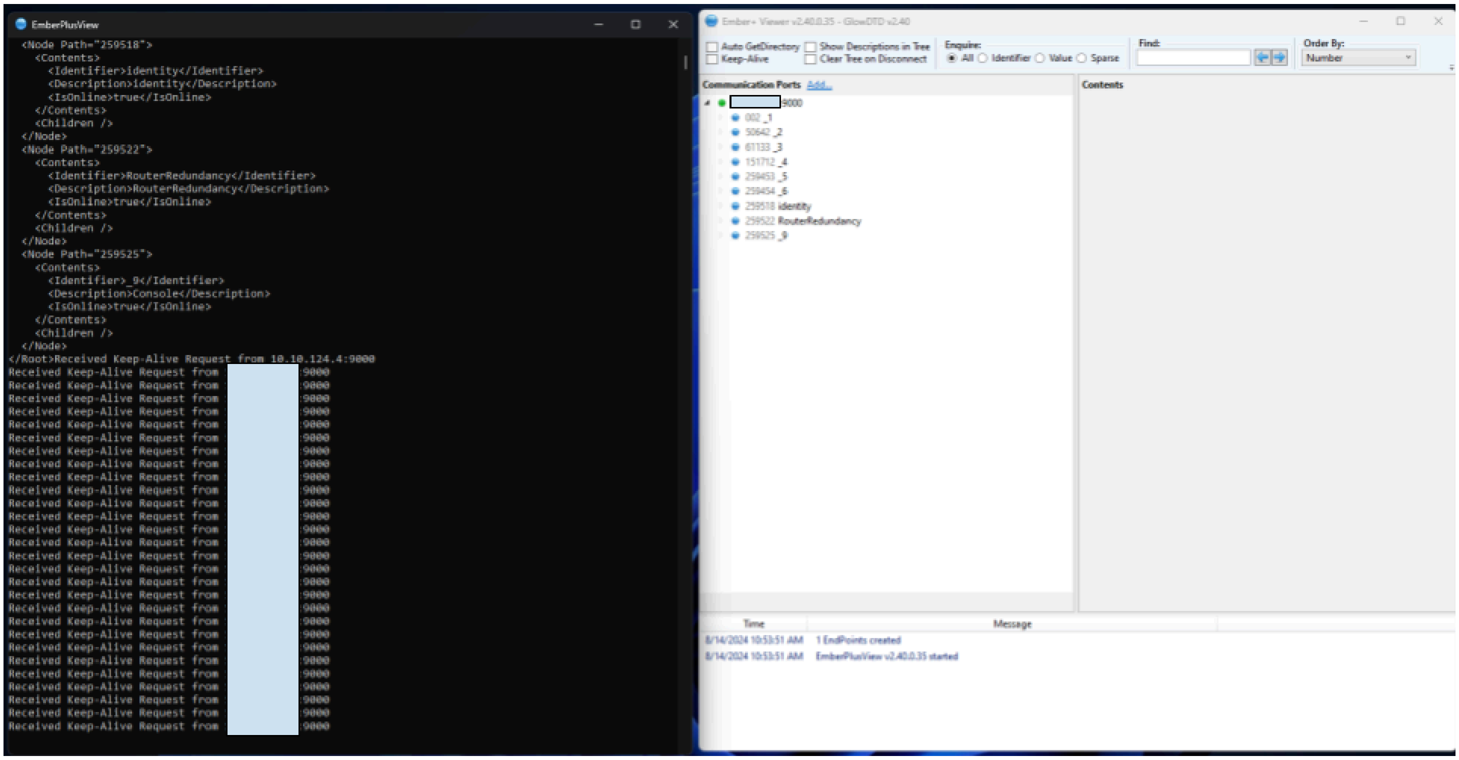
RemoteMNOPL is an older command protocol, not widely used anymore. It is not very comprehensive, offering only very basic crosspoint control of routers. It has been superseded by Ember+ in all respects and should only be used if for whatever reason Ember+ is not an option for you.

Side note; Cerebrum used RemoteMNOPL... Don't know why.

Emberplus;

Ember+ (emberplus) is a protocol partly designed by Lawo. It offers control of every aspect of mc². With the depth of the protocol comes some complexity in using it.

Lawo produces a tool for devices that use the Ember+ protocol - Ember+ Viewer. The software uses the Ember+ control protocol developed by Lawo to get access to the tree structure of Ember+ devices.



The software allows the user to connect to a Lawo device via it's IPv4 address, and inspect and manipulate all aspects of the device.

The software offers a GUI interface for manipulating aspects of the connected device, and a terminal output providing system information in an xml format.

Control systems that support Ember+ protocol do not need this software to connect to and control devices, Cerebrum for example, but it is a good tool for investigating and diagnosing Lawo systems.

Designing control systems with Emberlus

Note: The following method require the installation of Node.js with npm;

Windows users⤵

```
choco install nodejs.install
```

(or download and install from here -> (https://nodejs.org/en/download/prebuilt-installer)

Debian based users⤵

```
sudo apt install nodejs
sudo apt install npm
```

Most devices using emberplus run as a server.

A control system using emberplus to communicate will run as a client.

When running an Emberplus command, first start an Ember client, get the directory you wish to make a change to, issue the command, then disconnect from the directory.

This is an example of a Node.js function to do the above;

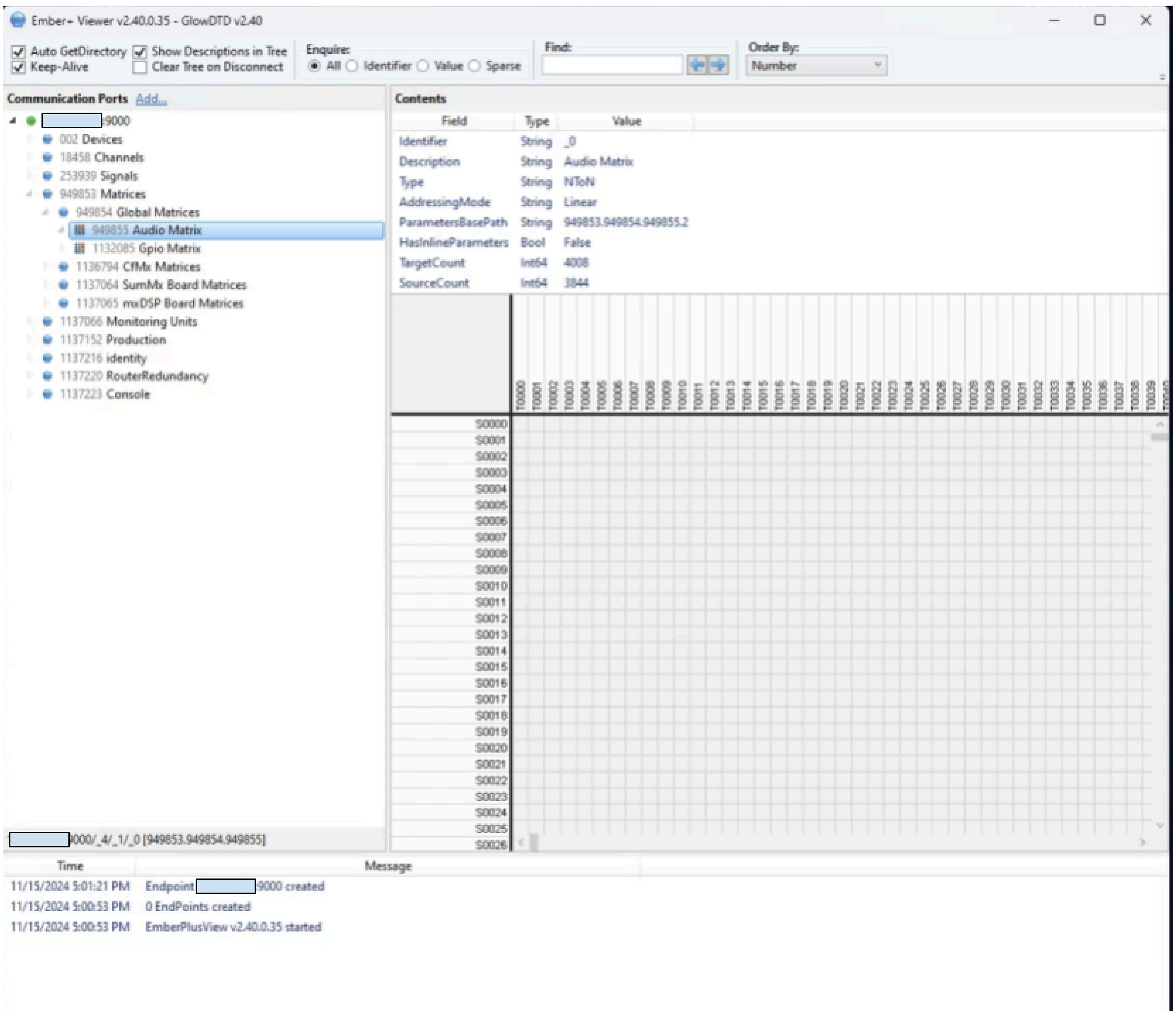
Node.js

```
async function runClient() { // Run an Ember Client
    const { host, port, source, target } = argv;
    const client = new EmberClient({ host, port, logger: new LoggingService(5) });

    client.on(EmberClientEvent.ERROR, e => {
        console.error('Error:', e);
    });

    try {
        await client.connectAsync();
        console.log(`Connected to Lawo MC² console at ${host}:${port}`);

        await client.getDirectoryAsync();
        let matrix = await client.getElementByPathAsync("_4/_1/_0"); // Get the directory you
want to make a change in

        console.log(`Connecting source ${source} to target ${target}`); // Issue a command
        await client.matrixConnectAsync(matrix, target, [source]);

    } catch (e) {
        console.error('Error:', e.stack);
    } finally {
        await client.disconnectAsync(); // Disconnect from the directory
        console.log('Disconnected from Lawo MC² console');
    }
}
```

The above is an example of controlling Lawo mc²/ Nova using node-emberplus ->(https://github.com/bmayton/node-emberplus)

Crosspoint Control

To make a crosspoint command you must know the source (source) and destination (target) IDs. You can get these using Emberplus Viewer (https://github.com/Lawo/ember-plus/releases/)
Below is an example of the location of the Audio Matrix on a Lawo mc² console.
Note the Node address (_04/_1/_0)



Command examples;

Here we will dive into some examples of control scripts for mc2 using emberplus.

All examples below are node.js scripts with command line options to target a specific router.

In the bellow examples specific ip-address, source-id, destination-id, have been omitted, to be replaced with the specific information from your router. See above about using Emberplus Viewer to obtain the information.

Connect crosspoint

Commandline⤵

```
node mc2Connect.js -h *IP-ADDRESS* -p 9000 -s *SOURCE-ID* -t *DESTINATION-ID*
```

Node.js⤵

```
// mc2Connect.js
const { EmberClient, EmberClientEvent, LoggingService } = require('node-emberplus');
const yargs = require('yargs/yargs');
const { hideBin } = require('yargs/helpers');

const argv = yargs(hideBin(process.argv))
    .option('host', {
        alias: 'h',
        type: 'string',
        demandOption: true,
        description: 'Lawo MC² host IP address',
    })
    .option('port', {
        alias: 'p',
        type: 'number',
        demandOption: true,
        description: 'Lawo MC² port',
    })
    .option('source', {
        alias: 's',
        type: 'number',
        demandOption: true,
        description: 'Source node to connect',
    })
    .option('target', {
        alias: 't',
```

```
        type: 'number',
        demandOption: true,
        description: 'Target node to connect',
    })
    .help()
    .argv;

async function runClient() {
    const { host, port, source, target } = argv;  // Use command-line arguments for host, port, source, and target
    const client = new EmberClient({ host, port, logger: new LoggingService(5) });

    client.on(EmberClientEvent.ERROR, e => {
        console.error('Error:', e);
    });

    try {
        await client.connectAsync();
        console.log(`Connected to Lawo MC² console at ${host}:${port}`);

        await client.getDirectoryAsync();
        let matrix = await client.getElementByPathAsync("_4/_1/_0"); // Check if this the Audio Martix of your console, it was for both of mine, so it might be a global destination...

        console.log(`Connecting source ${source} to target ${target}`);
        await client.matrixConnectAsync(matrix, target, [source]);

    } catch (e) {
        console.error('Error:', e.stack);
    } finally {
        await client.disconnectAsync(); // Close the connection
        console.log('Disconnected from Lawo MC² console');
    }
}

runClient();
```

Disconnect crosspoint

Commandline↴

```
node mc2Disconnect.js -h *IP-ADDRESS* -p 9000 -s 0 -t *DESTINATION-ID*
```

(note source '0' is used to disconnect)

Node.js

```
// mc2Disconnect.js
const { EmberClient, EmberClientEvent, LoggingService } = require('node-emberplus');
const yargs = require('yargs/yargs');
const { hideBin } = require('yargs/helpers');

const argv = yargs(hideBin(process.argv))
    .option('host', {
        alias: 'h',
        type: 'string',
        demandOption: true,
        description: 'Lawo MC² host IP address',
    })
    .option('port', {
        alias: 'p',
        type: 'number',
        demandOption: true,
        description: 'Lawo MC² port',
    })
    .option('source', {
        alias: 's',
        type: 'number',
        demandOption: true,
        description: 'Source node to connect',
    })
    .option('target', {
        alias: 't',
        type: 'number',
        demandOption: true,
        description: 'Target node to connect',
    })
    .help()
    .argv;

async function runClient() {
    const { host, port, source, target } = argv;
    const client = new EmberClient({ host, port, logger: new LoggingService(5) });

    client.on(EmberClientEvent.ERROR, e => {
        console.error('Error:', e);
    });

    try {
```

```
        await client.connectAsync();
        console.log(`Connected to Lawo MC² console at ${host}:${port}`);

        let matrix = await client.getElementByPathAsync("_4/_1/_0");

        console.log(`Disconnecting source ${source} to target ${target}`);
        await client.matrixDisconnectAsync(matrix, target, [source]);

    } catch (e) {
        console.error('Error:', e.stack);
    } finally {
        await client.disconnectAsync(); // Close the connection
        console.log('Disconnected from Lawo MC² console');
    }
}

runClient();
```

The below example would be used to mute a particular fader on an mc2.
Note this function would need to be adjusted to;
-Insert IP Address
-Port is assumed to be 9000 (standard emberplus port)
-ElementPath (get this from Emberplus Viewer)

Node.js

```
// MuteFader.js
const { EmberClient, EmberClientEvent, Emberlib, LoggingService } =
require('node-emberplus');

async function runClient() {
    const host = 'IP-ADDRESS'; // Change this to the consoles IP
    const port = 9000; // Standard Emberplus port is 9000
    const client = new EmberClient({ host, port, logger: new LoggingService(5) });

    client.on(EmberClientEvent.ERROR, e => {
        console.error('Error:', e);
    });

    try {
```

```
        await client.connectAsync();
        console.log(`Connected to Lawo MC² console at ${host}:${port}`);

        // Get the Mute parameter node
        let muteNode = await client.getElementByPathAsync("_3/_1/_4/_d00/_e/_400016c0/_400016c1"); // This is the ElementPath of the mute parameter of a particular fader, get your own
        console.log('Mute node:', muteNode);

        // Check if muteNode is a parameter, and then set its value
        if (muteNode && muteNode.contents && muteNode.contents.value !== undefined) {
            await client.setValueAsync(muteNode, false); // Set a new mute value (true/false)
            console.log('Mute value set to true');


    } catch (e) {
        console.error('Error:', e.stack);
    } finally {
        await client.disconnectAsync(); // Close the connection
        console.log('Disconnected from Lawo MC² console');
    }
}

runClient();
```

VPro

The same function can be used to control all aspects of a Lawo VPro as well. The next example is a routing function from a VPro input to a VPro output.

Commandline

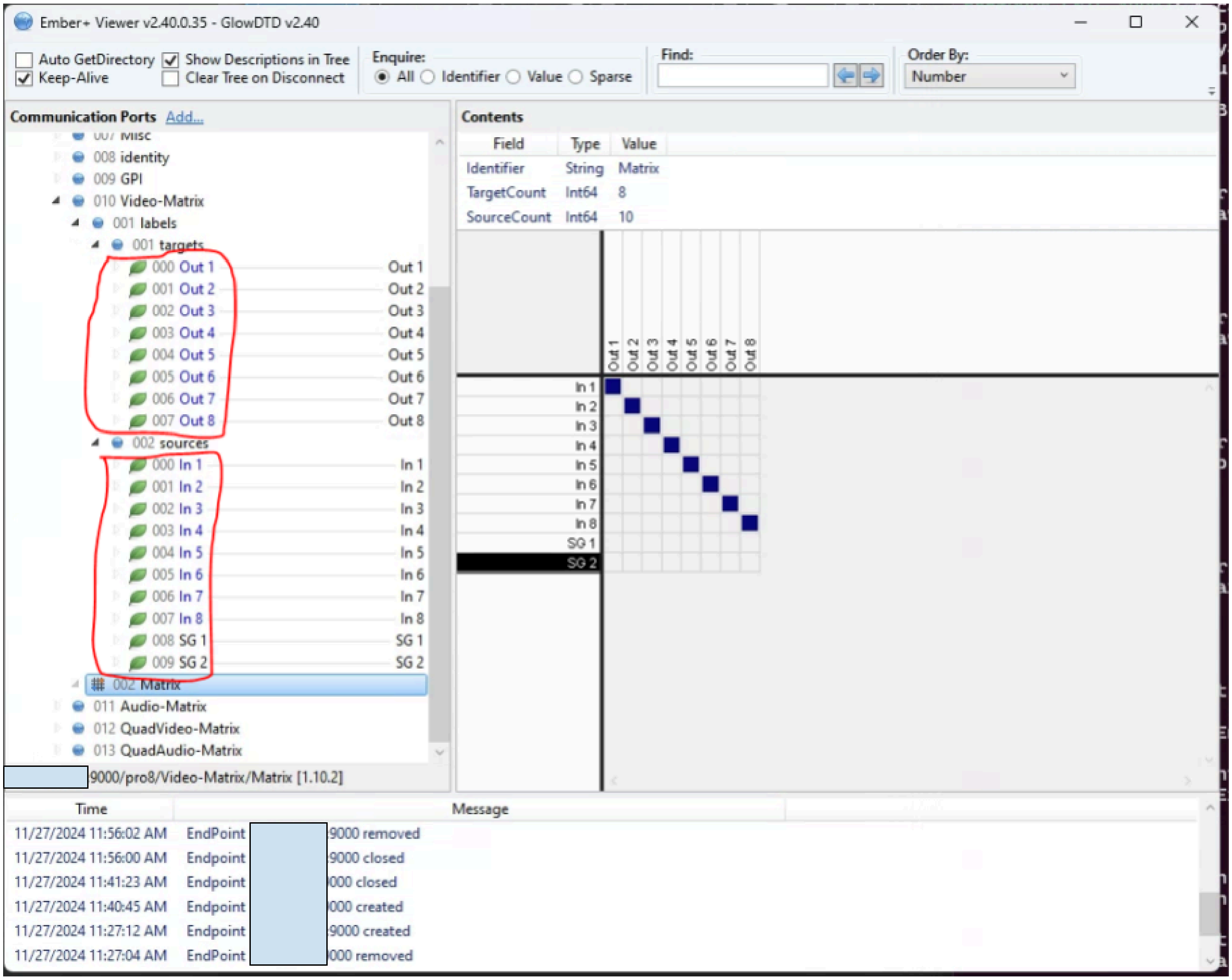
```
node ember_VPRO.js -h *IP-ADDRESS* -p 9000 -s 005 -t 006
```

Node.js

```
// ember_VPRO.js
const { EmberClient, EmberClientEvent, LoggingService } = require('node-emberplus');
const yargs = require('yargs/yargs');
const { hideBin } = require('yargs/helpers');

const argv = yargs(hideBin(process.argv))
    .option('host', {
        alias: 'h',
        type: 'string',
        demandOption: true,
        description: 'Lawo VPRO host IP address',
    })
    .option('port', {
        alias: 'p',
        type: 'number',
        demandOption: true,
        description: 'Lawo VPRO port',
    })
    .option('source', {
        alias: 's',
        type: 'number',
        demandOption: true,
        description: 'Source to connect',
    })
    .option('target', {
        alias: 't',
        type: 'number',
        demandOption: true,
        description: 'Target to connect',
    })
    .help()
    .argv;

async function runClient() {
    const { host, port, source, target } = argv;
    const client = new EmberClient({ host, port, logger: new LoggingService(5) });
```

```
    client.on(EmberClientEvent.ERROR, e => {
        console.error('Error:', e);
    });

    try {
        await client.connectAsync();
        console.log(`Connected to Lawo VPRO at ${host}:${port}`);

        await client.getDirectoryAsync();
        let matrix = await client.getElementByPathAsync("pro8/Video-Matrix/Matrix");

        console.log(`Connecting source ${source} to target ${target}`);
        await client.matrixConnectAsync(matrix, target, [source]);

    } catch (e) {
        console.error('Error:', e.stack);
    } finally {
        await client.disconnectAsync(); // Close the connection
        console.log('Disconnected from VPRO 3');
    }
}

runClient();
```

Not VPRO starts counting sources and destinations (targets) from 000, so 'In 5' = source '004'



This is the Matrix view from Ember+ Viewer, where we can see the label matrix identifiers.

Targets;
000 = Out 1
001 = Out 2
002 = Out 3
...

Sources;
000 = In 1
001 = In 2

```
002 = In 3
.....
```

3.8 AJA KiPro

AJA KiPro can be controlled via REST api assuming your KiPro is on the network.

Go to your KiPros's IP with the /descriptors.html endpoint (http://127.0.0.2/descriptors.html/) to get a full api list.

Some examples of useful commands;

Get recording state;

```
curl "http://xx.xx.xx.xx/config?action=get&paramid=eParamID_TransportState"
```

Set recording state to record;

```
curl "http://xx.xx.xx.xx/config?action=set&paramid=eParamID_TransportCommand&value=3"
```

Set recording state to stop;

```
curl "http://xx.xx.xx.xx/config?action=set&paramid=eParamID_TransportCommand&value=4"
```

Set recording state to play;

```
curl "http://xx.xx.xx.xx/config?action=set&paramid=eParamID_TransportCommand&value=1"
```

Set recording state to next clip;

```
curl "http://xx.xx.xx.xx/config?action=set&paramid=eParamID_TransportCommand&value=9"
```

Set recording state to previous clip;

```
curl "http://xx.xx.xx.xx/config?action=set&paramid=eParamID_TransportCommand&value=10"
```

Get amount of storage remaining;

```
curl "http://xx.xx.xx.xx/config?action=get&paramid=eParamID_CurrentMediaAvailable"
```

Get name of current clip;

```
curl "http://xx.xx.xx.xx/config?action=get&paramid=eParamID_CurrentClip"
```

Feedback example;

If you were to enquire the KiPro's recording state using;

```
curl "http://xx.xx.xx.xx/config?action=get&paramid=eParamID_TransportState"
```

... the response would look like this;

```
{"paramid":"2097217802","name":"eParamID_TransportState","value":"1","value_name":"Idle"}
```

...with the "value_name" being the key parameter to parse, in this example "Idle".

3.9 TSL-UMDv5

TSL-UMSv5 is a protocol used to send data to devices in a serial like fashion. It is commonly used to update Under Monitor Displays (UMD) and continues to be used to set values to multiple destinations hosted on a single device.

We use TSL-UMDv5 to update multiviews (Imagine Platinum), change studio/pcr states (IDS) and get tally information from vision mixers, just to name a few.

Below I will demonstrate a template TSL-UMDv5 node.js script and the parameters that can be used to set states on a few devices.

Note: TSL-UMDv5 is a serial like command, and as such can be problematic when this data is being streamed by two devices. Unlike sending a route command to a router, which is handled as a one-shot request, TSL-UMDv5 tends to be favoured as a periodic update command to end devices. This means that your control system will set a value, and periodically send the value to the device.
This is important to understand as if using the below commands in conjunction with an existing control system (say, Cerebrum) you will constantly be competing with the data being streamed from Cerebrum. Any updates you send to a device can be replaced with data being streamed for Cerebrum at its next polling interval. With that said, this is how to construct TSL-UMDv5 commands.

An example of using node.js to send TSL-UMDv5 data to update UMD text at a multiview;

Useage;

```
node umd-label.js -h xx.xx.xx.xx -p 4003 -w 15 -l "STUDIO 1"
```

The above command options are;
-h IP address of the multiview card
-p Port used by the multiview card for UMD data
-w The window (or PIP) you wish to update
-l The label you want to use for the update

NOTE: IP3 Multiview is index 0, so PIP-1 = Window 0.
This indexing is taken care of in the script, so use PIP number, not window number.

Node.js↴

```
#!/usr/bin/env node
const TSL5 = require('tsl-umd-v5');
```

```javascript
// Parse command line arguments
const args = process.argv.slice(2);
let host = null;
let displayAddress = null;
let label = null;
let port = null;

// Process arguments
for (let i = 0; i < args.length; i++) {
    switch (args[i]) {
        case '-h':
            host = args[++i];
            break;
        case '-p':
            port = parseInt(args[++i]);
            if (isNaN(port) || port < 1 || port > 65535) {
                console.error('Port must be between 1 and 65535');
                process.exit(1);
            }
            break;
        case '-w':
            displayAddress = parseInt(args[++i]);
            if (isNaN(displayAddress) || displayAddress < 1 || displayAddress > 1024) {
                console.error('Display address must be between 1 and 1024');
                process.exit(1);
            }
            break;
        case '-l':
            label = args[++i];
            break;
        default:
            console.error('Unknown option:', args[i]);
            console.error('Usage: node script.js [-h host] [-w display_address] [-l label] [-p port]');
            process.exit(1);
    }
}

if (!label) {
    console.error('Label is required. Use -l to specify a label');
    console.error('Usage: node script.js [-h host] [-w display_address] [-l label] [-p port]');
    process.exit(1);
}

// Initialize TSL UMD instance
const umd = new TSL5();
```

```javascript
// Create a tally object
const tally = {
    screen: 0,
    index: displayAddress - 1,  // Convert from 1-based to 0-based indexing
    display: {
        rh_tally: 0,
        text_tally: 0,
        lh_tally: 0,
        brightness: 3,
        text: label
    }
};

console.log(`Setting UMD label for display ${displayAddress} (index ${displayAddress - 1}) on
${host}:${port} to "${label}"`);

// Send both TCP and UDP
Promise.all([
    new Promise((resolve) => {
        try {
            umd.sendTallyTCP(host, port, tally);
            console.log('TCP message sent successfully');
            resolve();
        } catch (err) {
            console.error('TCP Error:', err);
            resolve();
        }
    }),
    new Promise((resolve) => {
        try {
            umd.sendTallyUDP(host, port, tally);
            console.log('UDP message sent successfully');
            resolve();
        } catch (err) {
            console.error('UDP Error:', err);
            resolve();
        }
    })
]).then(() => {
    // Give time for messages to be sent before exiting
    setTimeout(() => {
        console.log('Done.');
        process.exit(0);
    }, 200);
});
```

The same command can be used to send updates to IDS for example by modifying the command prompt;

Useage;

```
node umd-label.js -h xx.xx.xx.xx -p 2000 -w 10 -l "1"
```

The above command options are;
-h IP address of the IDS server
-p Port used for tsl data by IDS
-w The window (or display) you wish to update
-l The numeric value assigned to the state you wish to activate.

In this above example, VO4 (window 10) is put in state '1' (on-air)

Notes: Whilst writing this, I noticed that both Imagine and IDS build their device lists with index 0 (counting from 0 instead of 1), so this forms part of the script. I haven't come across a tsl-umd device that is index 1 yet, but if you are using this script on an index 1 device, you will need to manually accommodate the index 0 nature of the script by offsetting your command, or modify the script.

3.10 SAM Kahuna Tally

Tally information can be easily obtained form SAM Kahunaa as it is broadcast form the switcher. Running nc on the port usually allows you to view the tally information of a device on the network.
Let's look at Sam Kahunna for example. If we run 'nc xx.xx.xx.xx 50009' in a terminal, we will see the raw tally information being broadcast from the Sam Kahuna;

```
$ nc xx.xx.xx.xx 50009
'STOR 8STOR 9'  SUB CLIP 24''PGM'@PVW'CLEAN 'ME1 OP4'ME2 PGM'ME2 OP2'ME2 OP3'ME2 OP4'ME2 OP5'ME2 OP6'
                    @CCU 13'FEXT-13''ME3 PGM^C
```

As you will notice however, there is no discernible formatting, and this output will be hard to work with.

SAM Kahnuna also has multiple tally levels;
1 - PGM
2 - PVW
3 - ISO 1
4 - ISO 2
5 - ISO 3
... all of which will present in the above output as equal values, not helpful.

We handle this data better however.

Commandline

```
node SAM_Kahuna_tally.js -h IP-ADDRESS -p PORT
```

Node.js

```
#!/usr/bin/env node
const net = require('net');
// Configuration
const args = process.argv.slice(2);
let host = null;
let port = null;
// List of sources to ignore
const IGNORED_SOURCES = new Set([
    'PGM',
    'PVW',
    'ME2 PGM',
```

```
    'ME3 PGM',
    'STOR 1',
    'STOR 5'
]);
// Process arguments
for (let i = 0; i < args.length; i++) {
    switch (args[i]) {
        case '-h':
            host = args[++i];
            break;
        case '-p':
            port = parseInt(args[++i]);
            break;
        case '--debug':
            process.env.DEBUG = 'true';
            break;
        case '--help':
            console.log('Usage: node script.js [-h host] [-p port] [--debug]');
            process.exit(0);
    }
}
function cleanSourceName(text) {
    return text
        .replace(/^\x00+/, '')      // Remove leading null bytes
        .replace(/\x00+$/, '')      // Remove trailing null bytes
        .trim();                    // Remove any whitespace
}
function interpretTallyStatus(controlByte, sourceName) {
    // Clean the source name
    const cleanName = cleanSourceName(sourceName);

    if (process.env.DEBUG) {
        console.log('=======================================');
        console.log(`Raw source: ${cleanName}`);
        console.log(`In ignore list?: ${IGNORED_SOURCES.has(cleanName)}`);
        console.log(`Control byte: 0x${controlByte.toString(16).toUpperCase()}`);
        console.log('=======================================');
    }

    // Check against ignore list
    if (IGNORED_SOURCES.has(cleanName)) {
        return null;
    }
    switch (controlByte) {
        case 0x90:
        case 0xA0:
            return `${cleanName}`;
```

```
        default:
            return null;
    }
}
function parseKahunaTally(data) {
    const messages = [];
    const chunkSize = 24;
    for (let offset = 0; offset < data.length; offset += chunkSize) {
        if (offset + chunkSize <= data.length) {
            const chunk = data.slice(offset, offset + chunkSize);
            const controlByte = chunk[8];
            const text = chunk.slice(10, 24).toString('ascii');
            const status = interpretTallyStatus(controlByte, text);
            if (status) {
                messages.push(status);
            }
        }
    }
    return messages;
}
const client = new net.Socket();
client.connect(port, host);
client.on('data', (data) => {
    const messages = parseKahunaTally(data);
    messages.forEach(msg => console.log(msg));
});
client.on('error', (err) => {
    console.error('Connection error:', err);
});
// Handle graceful shutdown
process.on('SIGINT', () => {
    client.destroy();
    process.exit(0);
});
```

Using the above script, we can use the raw hex output form the port broadcasting the tally messages, and extract the control byte, which will indicate which tally level the source is routed to, and also exclude sources we are not interested in.

The resulting output looks like this;

```
$ node KahunaTally.js -h xx.xx.xx.xx -p 50009
PGM - VPR1.1
PGM - EXT-12
```

```
PGM - CCU 12
PGM - VIZ 1-1
PGM - VIZ 1-1
PGM - CCU 2
PGM - VPR2.1
...
```

As there are two switchers (vision mixers) using the same Kahunna, we have two control values to determining the PCR1/PCR2 PGM signal;

- 0x90
- 0xA0

Other control bytes used by SAM Kahnunna for pvw are (I think);

- 0x40
- 0xC0

ISO 1-3 control bytes are (I think);

- 0Xe0
- 0Xd0
- As well as some others I have not worked out...

Use '--debug' option to view tally output with control bytes visible;

Commandline

```
node SAM_Kahuna_tally.js -h IP-ADDRESS -p PORT --debug
```

```
========================================
Raw source: VPR1.2
In ignore list?: false
Control byte: 0x90
========================================
========================================
Raw source: VPR2.2
In ignore list?: false
Control byte: 0x40
========================================
VPR1.2
========================================
Raw source: CCU 12
In ignore list?: false
Control byte: 0x90
```

```
=======================================
=======================================
Raw source: VPR2.2
In ignore list?: false
Control byte: 0x40
=======================================
CCU 12
=======================================
Raw source: VPR1.2
In ignore list?: false
Control byte: 0x90
=======================================
=======================================
Raw source: VPR2.2
In ignore list?: false
Control byte: 0x40
=======================================
VPR1.2
=======================================
Raw source: VPR1.4
In ignore list?: false
Control byte: 0x0
=======================================
=======================================
Raw source: CCU 12
In ignore list?: false
Control byte: 0x90
=======================================
=======================================
Raw source: VPR1.2
In ignore list?: false
Control byte: 0x40
=======================================
CCU 12
```