README

Perry Kundert

August 18, 2014

Contents

1 Comm. Protocol Python Parser and Originator

Cpppo (pronounced 'c'+3*'p'+'o' in Python) is used to implement binary communications protocol parsers. The protocol's communication elements are described in terms of state machines which change state in response to input events, collecting the data and producing output data artifacts.

1.1 Installing

Cpppo depends on several Python packages:

Package	For?	Description
greenery	all	Regular Expression parsing and state machinery library
web	web API	The web.py HTTP web application framework
pytz	history	The Python time-zone library
tzlocal	history	Access to system's local timezone (on Mac, Windows)
pymodbus	remote	Modbus/TCP support for polling Schneider compatible PLCs
pytest	unit test	A Python unit-test framework

To install 'cpppo' and its required dependency 'greenery' using pip:

```
$ pip install cpppo
```

To install all required and optional Python modules, use:

```
pip install -r requirements.txt
pip install -r requirements-optional.txt
```

If you need system admin privileges to install packages, you may need to use 'sudo' on the above pip commands:

```
$ sudo pip install ...
```

1.1.1 Installing from source

Clone the repo by going to your preferred source directory and using:

```
$ git clone git@github.com:pjkundert/cpppo.git
```

You can then install from the provided setuptools-based setup.py installer:

```
$ cd cpppo
$ python setup.py install
```

1.1.2 Python Version Support

Cpppo is implemented and fully tested on both Python 2 and 3. However, some of cpppo's modules are not (yet) fully supported in both versions:

- The pymodbus module does not support Python 3, so Modbus/TCP support for polling remote PLCs is only available for Python 2.
- Greenery supports both Python 2 and 3, but doesn't provide meaningful Unicode (UTF-8) support in Python 2, so regular expression based DFAs are only supported for Python 3.

2 Protocols

The protocols implemented (presently, few) are described here.

2.1 EtherNet/IP CIP Logix Controller Simulation

A subset of the EtherNet/IP client and server protocol is implemented, and a simulation of a very small subset of a *Logix Controller is provided. It is capable of simulating Logix "tags", and the Read/Write Tag [Fragmented] services.

Only EtherNet/IP "Unconnected" type connections are supported. These are (anomalously) persistent connections to a single EtherNet/IP device (such as a Controller), which allow a sequence of CIP service requests (commands) to be sent to arbitrary CIP objects resident on the target device.

2.1.1 EtherNet/IP Controller Simulator

To run a simulation of a small subset of a Logix(tm) Controller with the tag 'SCADA' for you to read/write, run:

```
$ python -m cpppo.server.enip -v SCADA=INT[1000]
# (alternatively, invoke enip_server <options> ... )
```

These options are available when you execute the cpppo.server.enip module:

Specify a different local interface and/or port to bind to (default is 0.0.0.0:44818):

```
-a|--address [<interface>][:<port>]
```

Change the verbosity (supply more to increase further):

```
-v[vv...]|--verbose
```

Specify a constant or variable delay to apply to every response, in fractional seconds:

```
-d|--delay #.#[-#.#]
```

Specify an HTTP web server interface and/or port, if a web API is desired (just ':' will enable the web API on defaults 0.0.0.0:80, or whatever interface was specified for –address):

```
-w|--web [<interface>]:[<port>]
```

To send log output to a file (limited to 10MB, rotates through 5 copies):

```
-l|--log <file>
```

To print a summary of PLC I/O to stdout:

```
-p|--print
```

You may specify as many tags as you like on the command line; at least one is required:

```
<tag>=<type>[<length>]   # eg. SCADA=INT[1000]
```

The available types are INT (16-bit), SINT (8-bit) DINT (32-bit) integer, and REAL (32-bit float).

2.1.2 EtherNet/IP Controller I/O Customization

If you require access to the data streaming from client to and from the EtherNet/IP CIP Attributes hosted in your simulated controller, you can easily provide a custom cpppo.server.enip.device Attribute implementation which will receive all PLC Read/Write Tag [Fragmented] request data.

For example purposes, we have implemented the cpppo.server.enip.historize module which intercepts all I/O (and exceptions) and writes it to the file specified in the first command-line argument to the module:

```
$ python -m cpppo.server.enip.historize some_file.hst Tag_Name=INT[1000] &
$ tail -f some_file.txt
# 2014-07-15 22:03:35.945: Started recording Tag: Tag_Name
2014-07-15 22:03:44.186 ["Tag_Name", [0, 3]]    {"write": [0, 1, 2, 3]}
...
```

(in another terminal)

```
$ python -m cpppo.server.enip.client Tag_Name[0-3]=[0,1,2,3]
```

You can examine the code in cpppo/server/enip/historize.py to see how to easily implement your own customization of the EtherNet/IP CIP Controller simulator.

If you invoke the 'main' method provided by cpppo.server.enip.main directly, all command-line args will be parsed, and the EtherNet/IP service will not return control until termination. Alternatively, you may start the service in a separate threading.Thread and provide it with a list of configuration options. Note that each individual EtherNet/IP Client session is serviced by a separate Thread, and thus all method invocations arriving at your customized Attribute object need to process data in a Thread-safe fashion.

2.1.3 EtherNet/IP Controller Client

A simple EtherNet/IP Controller Client is provided. Presently, it only knows how to Register, and issue Read Tag Fragmented requests (and only for offset 0; the first block of results):

```
python -m cpppo.server.enip.client SCADA[1] SCADA[0-10] ...
# (alternatively, invoke enip_client <options> ... )
```

Specify a different local interface and/or port to connect to (default is 0.0.0.0:44818):

```
-a|--address [<interface>][:<port>]
```

Change the verbosity (supply more to increase further):

```
-v[vv...]|--verbose
```

Change the default response timeout

```
-t|--timeout #
```

Specify a number of times to repeat the specified operations:

```
-r|--repeat #
```

To send log output to a file (limited to 10MB, rotates through 5 copies):

```
-l|--log <file>
```

To print a summary of PLC I/O to stdout:

```
-p|--print
```

To force use of the Multiple Service Packet request, which carries multiple Read/Write Tag [Fragmented] requests in a single EtherNet/IP CIP I/O operation (default is to issue each request as a separate I/O operation):

```
-m|--multiple
```

To force the client to use plain Read/Write Tag commands (instead of the Fragmented commands, which are the default):

```
-n|--no-fragment
```

You may specify as many tags as you like on the command line; at least one is required. An optional register (range) can be specified (default is register 0):

```
<tag> <tag>[<reg>] <tag>[<reg>-<reg>] # eg. SCADA SCADA[1] SCADA[1-10]
```

Writing is supported; the number of values must exactly match the data specified register range:

```
<tag>=<value>                              # scalar, eg. SCADA=1
<tag>[<reg>-<reg>]=<value>,<value>,...     # vector range
<tag>[<reg>]=<value>                       # single element of a vector
<tag>[<reg>-<reg>]=(DINT)<value>,<value> # cast to SINT, INT, DINT or REAL
```

If any <value> contains a '.' (eg. '9.9,10'), all values are deemed to be REAL; otherwise, they are integers and default to a type INT. To force a specific type (and limit the values to the appropriate value range), you may specify a "cast" to a specific type, eg. 'TAG[4-6]=(INT)1,2,3'. The types SINT, INT, DINT and REAL are supported.

2.1.4 Web Interface

The following actions are available via the web interface. It is designed to be primarily a REST-ful HTTP API returning JSON, but any of these requests may be made via a web browser, and a minimal HTML response will be issued.

Start a Logix Controller simulator on port 44818 (the default), with a web API on port 12345:

```
python -m cpppo.server.enip -v --web :12345 SCADA=INT[1000]
```

The api is simple: api/<group>/<match>/<command>/<value> . There are 3 groups: “options”, “tags” and “connections”. If you don’t specify <group> or <match>, they default to the wildard “*”, which matches anything.

So, to get everything, you should now be able to hit the root of the api with a browser at: http://localhost:12345/api, or with wget or curl:

```
$ wget -qO - http://localhost:12345/api
$ curl http://localhost:12345/api
```

and you should get something like:

```
$ curl http://localhost:12345/api
{
    "alarm": [],
    "command": {},
    "data": {
        "options": {
            "delay": {
                "value": 0.0
            }
        },
        "server": {
            "control": {
                "disable": false,
                "done": false,
                "latency": 5.0,
                "timeout": 5.0
            }
        },
        "tags": {
            "SCADA": {
            "attribute": "SCADA          INT[1000] == [0, 0, 0, 0, 0, 0,...]",
            "error": 0
            }
        }
    },
    "since": null,
    "until": 1371731588.230987
}
```

- options/delay/value
  To access or modify some specific thing in the matching object(s), add a <command> and <value>:

```
$ curl http://localhost:12345/api/options/delay/value/0.5
{
    "alarm": [],
    "command": {
        "message": "options.delay.value=u'0.5' (0.5)",
        "success": true
    },
    "data": {
        "options": {
            "delay": {
                "value": 0.5
            }
        }
    },
    "since": null,
    "until": 1371732496.23366
}
```

  It will perform the action of assigning the <value> to all of the matching <command> entities. In this case, since you specified a precise <group> "options", and <match> "delay", exactly one entity was affected: "value" was assigned "0.5". If you are running a test client against the simulator, you will see the change in response time.

  As a convenience, you can use /<value> or =<value> as the last term in the URL:

```
$ curl http://localhost:12345/api/options/delay/value/0.5
$ curl http://localhost:12345/api/options/delay/value=0.5
```

- api/options/delay/range
  If you've started the simulator with –delay=0.1-0.9 (a delay range), you can adjust this range to a new range, using:

```
$ curl http://localhost:12345/api/options/delay/range=0.5-1.5
```

You can cause it to never respond (in time), to cause future connection attempts to fail:

```
$ curl http://localhost:12345/api/options/delay/value=10.0
```

Or, if you've configured a delay range using –delay=#-#, use:

```
$ curl http://localhost:12345/api/options/delay/range=10.0-10.0
```

Restore connection responses by restoring a reasonable response timeout.

- api/server/control/done or disable
  To prevent any future connections, you can (temporarily) disable the server, which will close its port (and all connections) and await further instructions:

  ```
  $ curl http://localhost:12345/api/server/control/disable/true
  ```

  Re-enable it using:

  ```
  $ curl http://localhost:12345/api/server/control/disable/false
  ```

  To cause the server to exit completely (and of course, causing it to not respond to future requests):

  ```
  $ curl http://localhost:12345/api/server/control/done/true
  ```

- api/server/control/latency or timeout
  The default socket I/O blocking 'latency' is .1s; this is the time it may take for each existing connection to detect changes made via the web API, eg. signalling EOF via api/connections/eof/true. The 'timeout' on each thread responding defaults to twice the latency, to give the thread's socket I/O machinery time to respond and then complete. These may be changed, if necessary, if simulation of high-latency links (eg. satellite) is implemented (using other network latency manipulation software).

- api/tags/<tagname>/error
  To force all successful accesses to a certain tag (eg. SCADA) to return a certain error code, you can set it using:

```
$ curl http://localhost:12345/api/tags/SCADA/error=8
```

Restore it to return success:

```
$ curl http://localhost:12345/api/tags/SCADA/error/0
```

- api/tags/<tagname>/attribute[x]
  To access or change a certain element of a tag, access its attribute at a certain index (curl has problems with this kind of URL):

```
wget -qO -  http://localhost:12345/api/tags/SCADA/attribute[3]=4
```

You can access any specific value to confirm:

```
wget -qO -  http://localhost:12345/api/tags/SCADA/attribute[3]
{
    "alarm": [],
    "command": {
        "message": "tags.SCADA.attribute[2]: 0",
        "success": true
    },
    "data": {
        "tags": {
            "SCADA": {
                "attribute": "SCADA          INT[1000] == [0, 0, 0, 4, 0, 0,
                ...]",
                "error": 0
            }
        }
    },
    "since": null,
    "until": 1371734234.553135
}
```

- api/connections/*/eof
  To immediately terminate all connections, you can signal them that they've experienced an EOF:

```
$ curl http://localhost:12345/api/connections/*/eof/true
```

If there are any matching connections, all will be terminated. If you know the port and IP address of the interface from which your client is connecting to the simulator, you can access its connection specifically:

```
$ curl http://localhost:12345/api/connections/10_0_111_121_60592/eof/true
```

To wait for all connections to close, you can issue a request to get all connections, and wait for the 'data' attribute to become empty:

```
$ curl http://localhost:12345/api/connections
{
    "alarm": [],
    "command": {},
    "data": {
        "connections": {
            "127_0_0_1_52590": {
                "eof": false,
                "interface": "127.0.0.1",
                "port": 52590,
                "received": 1610,
                "requests": 17
            },
            "127_0_0_1_52591": {
                "eof": false,
                "interface": "127.0.0.1",
                "port": 52591,
                "received": 290,
                "requests": 5
            }
        }
    },
    "since": null,
    "until": 1372889099.908609
}
$ # ... wait a while (a few tenths of a second should be OK)...
$ curl http://localhost:12345/api/connections
{
    "alarm": [],
    "command": null,
    "data": {},
```

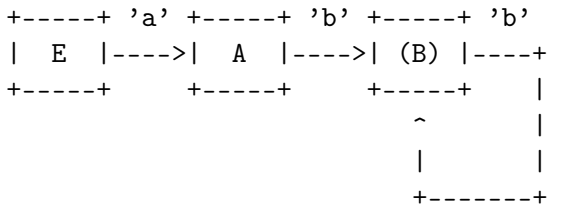
```
        "since": null,
        "until": 1372889133.079849
    }
```

3 Deterministic Finite Automata

A cpppo.dfa will consume symbols from its source iterable, and yield (machine,state) transitions 'til a terminal state is reached. If 'greedy', it will transition 'til we reach a terminal state and the next symbol does not product a transition.

For example, if 'abbb,ab' is presented to the following machine with a no-input state E, and input processing states A and (terminal) B, it will accept 'ab' and terminate, unless greedy is specified in which case it will accept 'abbb' and terminate.

3.1 Basic State Machines



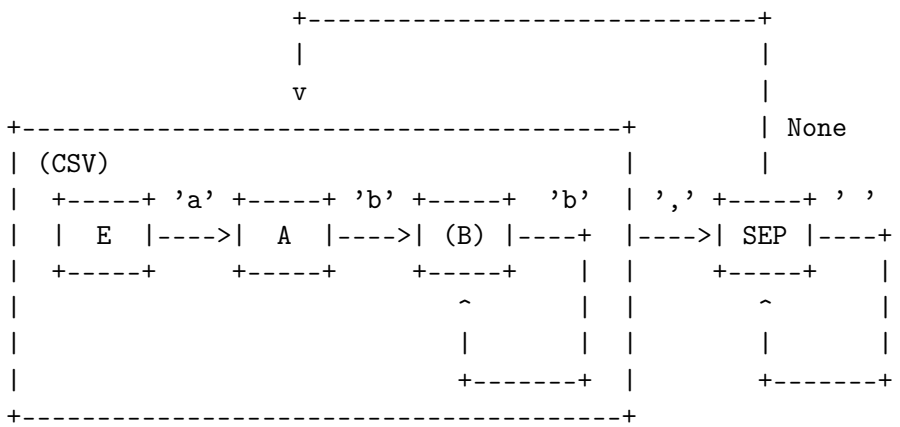
This machine is easily created like this:

```
# Basic DFA that accepts ab+
E                   = cpppo.state( "E" )
A                   = cpppo.state_input( "A" )
B                   = cpppo.state_input( "B", terminal=True )
E['a']              = A
A['b']              = B
B['b']              = B

BASIC               = cpppo.dfa( 'ab+', initial=E, context='basic' )
```

3.2 Composite Machines

A higher-level DFA can be produced by wrapping this one in a cpppo.dfa, and giving it some of its own transitions. For example, lets make a machine that accepts 'ab+' separated by ',[ ]*'.



This is implemented:

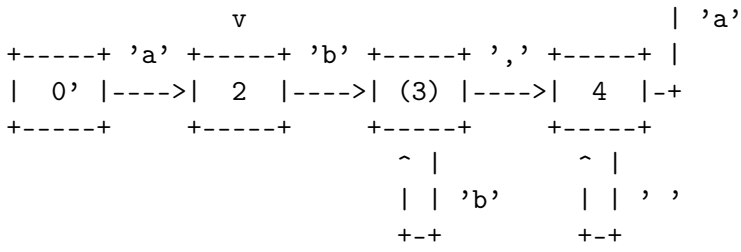
```
# Composite state machine accepting ab+, ignoring ,[ ]* separators
ABP                  = cpppo.dfa( "ab+", initial=E, terminal=True )
SEP                  = cpppo.state_drop( "SEP" )
ABP[',']             = SEP
SEP[' ']             = SEP
SEP[None]            = ABP

CSV                  = cpppo.dfa( 'CSV', initial=ABP, context='csv' )
```

When the lower level state machine doesn't recognize the input symbol for a transition, the higher level machine is given a chance to recognize them; in this case, a ',' followed by any number of spaces leads to a $\text{state}_{\text{drop}}$ instance, which throws away the symbol. Finally, it uses an "epsilon" (no-input) transition (indicated by a transition on None) to re-enter the main CSV machine to process subsequent symbols.

3.3 Machines from Regular Expressions

We use https://github.com/ferno/greenery to convert regular expressions into greenery.fsm machines, and post-process these to produce a cpppo.dfa. The regular expression '(ab+)((,[ ]*)(ab+))*' is equivalent to the above (except that it doesn't ignore the separators), and produces the following state machine:



A regular expression based cpppo.dfa is created thus:

```
# A regular expression; the default dfa name is the regular expression itself.
REGEX                = cpppo.regex( initial='(ab+)((,[ ]*)(ab+))*', context='regex' )
```

3.3.1 Unicode Support

Cpppo supports Unicode (UTF-8) on both Python 2 and 3. However, greenery provides meaningful Unicode support only under Python 3. Therefore, if you wish to use Unicode in regular expressions, you must use Python 3.

4 Running State Machines

State machines define the grammar for a language which can be run against a sentence of input. All these machines ultimately use state_input instances to store their data; the path used is the cpppo.dfa's <context> + '_input':

```
data                    = cpppo.dotdict()
for machine in [ BASIC, CSV, REGEX ]:
    path                = machine.context() + '.input' # default for state_input data
    source              = cpppo.peekable( str( 'abbbb, ab' ))
    with machine:
        for i,(m,s) in enumerate( machine.run( source=source, data=data )):
            print( "%s #%3d; next byte %3d: %-10.10r: %r" % (
                   m.name_centered(), i, source.sent, source.peek(), data.get(path) ))
    print( "Accepted: %r; remaining: %r\n" % ( data.get(path), ''.join( source )))
print( "Final: %r" % ( data ))
```

5 Virtualization

Software with an interface acting as a PLC is often deployed as an independent piece of infrastructure with its own IP address, etc. One simple

approach to do this is to use Vagrant to provision OS-level Virtualization resources such as VirtualBox and VMWare, and/or Docker to provision lightweight Linux kernel-level virtualizations.

Using a combination of these two facilities, you can provision potentially hundreds of "independent" PLC simulations on a single host – each with its own IP address and configuration.

5.1 Vagrant

If you are not running on a host capable of directly hosting Docker images, one can be provided for you. Install Vagrant (http://vagrantup.com) on your system, and then use the cpppo/GNUmakefile target to bring up a VirtualBox or VMWare Fusion (license required: http://www.vagrantup.com/vmware):

```
$ make vmware-debian-up # or virtualbox-ubuntu-up
```

Connect to the running virtual machine:

```
$ make vmware-debian-ssh
...
vagrant@jessie64:~$
```

Both Debian and Ubuntu Vagrantfiles are provided, which produce a VM image capable of hosting Docker images. Not every version is available on every platform, depending on what version of VMware or Virtualbox you are running; see the GNUmakefile for details.

5.1.1 Building a Vagrant Image

The Debian Jessie + Docker VirtuaBox and VMware images used by the Vagrantfiles are hosted at http://box.hardconsulting.com. When you use the cpppo/GNUmakefile targets to bring up a Vagrant box (eg. 'make virtualbox-debian-up'), the appropriate box is downloaded using 'vagrant box add ...'. If you don't trust these boxes (the safest position), you can rebuild them yourself.

- Packer.io
  Using the packer tool, build a VirtualBox (or VMware) image. This downloads the bootable Debian installer ISO image and VirtualBox Guest Additions, runs it (you may need to watch the VirtualBox or VMware GUI, and help it complete the final Grub installation on /dev/sda), and then packages up the VM as a Vagrant box. We'll rename it jessie64, and augment the zerodisk.sh script to flush its changes to the device:

```
$ cd src/cpppo/packer
$ make vmware-jessie64 # or virtualbox-jessie64
...
```

Once it builds successfully, add the new box to the ../docker/debian Vagrant installation, to make it accessible:

```
$ make add-vmware-jessie64 # or add-virtualbox-jessie64
```

Now, you can fire up the new VirtualBox image using Vagrant, and the targets provided in the cpppo/GNUmakefile:

```
$ cd src/cpppo
$ make vmware-debian-up
```

5.2 Docker

We'll assume that you now have a prompt on a Docker-capable machine. Start a Docker container using the pre-built cpppo/cpppo image hosted at https://index.docker.io/u/cpppo/. This will run the image, binding port 44818 on localhost thru to port 44818 on the running Docker image, and will run the cpppo.server.enip module with 1000 16-bit ints on Tag "SCADA":

```
$ docker run -p 44818:44818 -d cpppo/cpppo python -m cpppo.server.enip SCADA=dint[1
6da5183740b4
$
```

A canned Docker image is provided which automatically runs an instance of cpppo.server.enip hosting the "SCADA=dint[1]" tag by default (you can provide alternative tags on the command line, if you wish):

```
$ docker run -p 44818:44818 -d cpppo/scada
```

Assuming you have cpppo installed on your local host, you can now test this. We'll read a single value and a range of values from the tag SCADA, repeating 10 times:

```
$ python -m cpppo.server.enip.client -r 10 SCADA[1] SCADA[0-10]
10-08 09:40:29.327  ...  SCADA[    1-1    ] == [0]
10-08 09:40:29.357  ...  SCADA[    0-10   ] == [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

[1] DEFINITION NOT FOUND: 1000

```
10-08 09:40:29.378  ...  SCADA[    1-1    ] == [0]
10-08 09:40:29.406  ...  SCADA[    0-10   ] == [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
10-08 09:40:29.426  ...  SCADA[    1-1    ] == [0]
10-08 09:40:29.454  ...  SCADA[    0-10   ] == [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
10-08 09:40:29.476  ...  SCADA[    1-1    ] == [0]
10-08 09:40:29.503  ...  SCADA[    0-10   ] == [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
10-08 09:40:29.523  ...  SCADA[    1-1    ] == [0]
10-08 09:40:29.551  ...  SCADA[    0-10   ] == [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
10-08 09:40:29.571  ...  SCADA[    1-1    ] == [0]
10-08 09:40:29.600  ...  SCADA[    0-10   ] == [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
10-08 09:40:29.622  ...  SCADA[    1-1    ] == [0]
10-08 09:40:29.648  ...  SCADA[    0-10   ] == [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
10-08 09:40:29.669  ...  SCADA[    1-1    ] == [0]
10-08 09:40:29.697  ...  SCADA[    0-10   ] == [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
10-08 09:40:29.717  ...  SCADA[    1-1    ] == [0]
10-08 09:40:29.745  ...  SCADA[    0-10   ] == [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
10-08 09:40:29.769  ...  SCADA[    1-1    ] == [0]
10-08 09:40:29.796  ...  SCADA[    0-10   ] == [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
10-08 09:40:29.796  ...  Client ReadFrg. Average  20.266 TPS (  0.049s ea).
$
```

5.2.1 Creating Docker images from a Dockerfile

Get started by going to .../cpppo/docker/cpppo/cpppo/Dockerfile on your local machine. If you started a Vagrant VM from this directory (eg. make vmware-up), this is also mounted inside that machine /src/cpppo. Once there, have a look at docker/cpppo/cpppo/Dockerfile. If you go into that directory, you can re-create the Docker image:

```
$ cd /src/cpppo/docker/cpppo/cpppo
$ docker build -t cpppo/cpppo .
```

Or, lets use it as a base image for a new Dockerfile. Lets just formalize the command we ran previously so we don't have to remember to type it in. Create a new Dockerfile in, say, cpppo/docker/cpppo/scada/:

```
FROM        cpppo/cpppo
MAINTAINER  Whoever You Are "whoever@example.com"
EXPOSE      44818
# We'll always run this as our base command
ENTRYPOINT  [ "python",  "-m", "cpppo.server.enip" ]
```

```
# But we will allow this to be (optionally) overridden
CMD         [ "SCADA=dint[1000]" ]
```

Then, we can build and save the container under a new name:

```
docker build -t cpppo/scada .
docker run -p 44818
```

This is (roughly) what is implemented in docker/cpppo/scada/Dockerfile.