README

Perry Kundert

May 2, 2015

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 and OS Support

Cpppo is implemented and fully tested on both Python 2 (2.6 and 2.7), and Python 3 (3.3 and 3.4). The EtherNet/IP CIP protocol implementation is fully tested and widely used in both Python 2 and 3.

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.

Linux (native or Docker containerized), Mac and Windows OSs are supported. However, Linux or Mac are recommended for stability, performance and ease of use. If you need to use Windows, it is recommended that you install a usable Terminal application such as ConEmu.

2 Protocols

The protocols implemented are described here.

2.1 EtherNet/IP CIP Controller Communications Simulator/Client

A subset of the EtherNet/IP client and server protocol is implemented, and a simulation of a subset of the Tag communications capability of a Allen-Bradley ControlLogix 5561 Controller is provided. It is capable of simulating ControlLogix Tag access, via the Read/Write Tag [Fragmented] services.

Only EtherNet/IP "Unconnected" type connections are supported. These are (somewhat anomalously) a persistent connection 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.

A Tag is simply a shortcut to a specific EtherNet/IP CIP Object Instance and Attribute. Instead of the Client needing to know the specific Instance and Attribute numbers, the more easily remembered and meaningful Tag may be supplied in the request path.

2.1.1 EtherNet/IP Controller Communications Simulator

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

```
$ python -m cpppo.server.enip --print SCADA=INT[1000]
```

Alternatively, invoke the supplied wrapper script:

```
enip_server --print SCADA=INT[1000]
```

This is especially handy under Python 2.6, where you must manually specify the module entry point, eg. `cpppo.server.enip.__main__`.

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

Specify a different local interface and/or port to bind to (default is `:44818`, indicating all interfaces and port 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 :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 read and write I/O events streaming from client(s) to and from the EtherNet/IP CIP Attributes hosted in your simulated controller, you can easily make a custom cpppo.server.enip.device Attribute implementation which will receive all PLC Read/Write Tag [Fragmented] request data.

We provide two examples; one which records a history of all read/write events to each Tag, and one which connects each Tag to the current temperature of the city with the same name as the Tag.

- Record Tag History
  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.

- City Temperature Tag
  In this example, we intercept read requests to the Tag, and look up the current temperature of the city named with the Tag's name. This example is simple enough to include here (see `cpppo/server/enip/weather.py`):

```
import sys, logging, json
try:
    from urllib2 import urlopen
except ImportError:
    from urllib.request import urlopen

from cpppo.server.enip import device
from cpppo.server.enip.main import main

class Attribute_weather( device.Attribute ):
    def __getitem__( self, key ):
        try:
            url = "http://api.openweathermap.org/data/2.5/weather?units=metric&q=%
            weather = json.loads( urlopen( url ).read() )
            return [ weather['main']['temp'] ]
        except Exception as exc:
            logging.warning( "Couldn't get temperature for %s: %s", self.name, exc
            raise
```

```
    def __setitem__( self, key, value ):
        raise Exception( "Changing the weather isn't that easy..." )

sys.exit( main( attribute_class=Attribute_weather ))
```

By providing a specialized implementation of device.Attribute's `__getitem__` (which is invoked each time an Attribute is accessed), we arrange to query the city's weather at the given URL, and return the current temperature. Of course, `__setitem__` (which would be invoked whenever someone wishes to change the city's temperature) would have a much more complex implementation, the details of which are left as an exercise to the reader...

2.1.3 EtherNet/IP Controller Client

A simple EtherNet/IP CIP Client is provided. It can Register and issue a stream of "Unconnected" requests to the Controller, such as Read/Write Tag (optionally Fragmented) requests:

```
python -m cpppo.server.enip.client -v --print SCADA[1]=99 SCADA[0-10]
```

Alternatively, invoke the supplied wrapper script:

```
enip_client SCADA[1]=99 SCADA[0-10]
```

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

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

On Windows systems, you must specify an actual interface. For example, if you started the cpppo.server.enip simulator above (running on the all interfaces by default), use `--address localhost`.

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.

In addition to symbolic Tag addressing, numeric Class/Instance/Attribute addressing is available. A Class, Instance and Attribute address values are in decimal by default, but hexadecimal, octal etc. are available using escapes, eg. 26 == 0x1A == 0o49 == 0b100110:

```
@<class>/<instance>/<attribute>           # read a scalar, eg. @0x1FF/01/0x1A
@<class>/<instance>/<attribute>[99]=1     # write element, eg. @511/01/26=1
```

See further details of addressing `cpppo.server.enip.client`'s `parse_operations` below.

2.1.4 EtherNet/IP `cpppo.server.enip.client` API

Dispatching a multitude of EtherNet/IP CIP I/O operations to a Controller (with our without pipelining) is very simple. If you don't need to see the results of each operation as they occur, or just want to ensure that they succeeded, you can use `connector.process` (see `cpppo/server/enip/client/io.py`):

```
host                        = 'localhost'   # Controller IP address
port                        = address[1]    # default is port 44818
depth                       = 1             # Allow 1 transaction in-flight
multiple                    = 0             # Don't use Multiple Service Packet
fragment                    = False         # Don't force Read/Write Tag Fragmented
timeout                     = 1.0           # Any PLC I/O fails if it takes > 1s
printing                    = True          # Print a summary of I/O
tags                        = ["Tag[0-9]+16=(DINT)4,5,6,7,8,9", "@0x2/1/1", "Tag[3-5]"]

with client.connector( host=host, port=port, timeout=timeout ) as connection:
    operations              = client.parse_operations( tags )
    failures,transactions   = connection.process(
        operations=operations, depth=depth, multiple=multiple,
        fragment=fragment, printing=printing, timeout=timeout )

sys.exit( 1 if failures else 0 )
```

Try it out by starting up a simulated Controller:

```
$ python -m cpppo.server.enip Tag=DINT[10] &
$ python -m cpppo.server.enip.io
```

The API is able to "pipeline" requests – issue multiple requests on the wire, while simultaneously harvesting prior requests. This is absolutely necessary in order to obtain reasonable I/O performance over high-latency links (eg. via Satellite).

To use pipelining, create a `client.connector` which establishes and registers a CIP connection to a Controller. Then, produce a sequence of operations (eg, parsed from "Tag[0-9]+16=(DINT)5,6,7,8,9" or from numeric Class, Instance and Attribute numbers "@2/1/1"), and dispatch the requests using connector methods `.pipeline` or `.synchronous` (to access the details of the requests and the harvested replies), or `.process` to simply get a summary of I/O failures and total transactions.

More advanced API methods allow you to access the stream of I/O in full detail, as responses are received. To issue command synchronously use

connector.synchronous, and to "pipeline" the requests (have multiple requests issued and "in flight" simultaneously), use connector.pipeline (see cpppo/server/enip/client/thruput.py)

```
ap                          = argparse.ArgumentParser()
ap.add_argument( '-d', '--depth',    default=0, help="Pipelining depth" )
ap.add_argument( '-m', '--multiple', default=0, help="Multiple Service Packet size lim
ap.add_argument( '-r', '--repeat',   default=1, help="Repeat requests this many times"
ap.add_argument( '-a', '--address',  default='localhost', help="Hostname of target Cont
ap.add_argument( '-t', '--timeout',  default=None, help="I/O timeout seconds (default:
ap.add_argument( 'tags', nargs='+', help="Tags to read/write" )
args                        = ap.parse_args()

depth                       = int( args.depth )
multiple                    = int( args.multiple )
repeat                      = int( args.repeat )
operations                  = client.parse_operations( args.tags * repeat )
timeout                     = None
if args.timeout is not None:
    timeout                 = float( args.timeout )

with client.connector( host=args.address, timeout=timeout ) as conn:
    start                   = cpppo.timer()
    num,idx                 = -1,-1
    for num,(idx,dsc,op,rpy,sts,val) in enumerate( conn.pipeline(
            operations=operations, depth=depth,
            multiple=multiple, timeout=timeout )):
        print( "%s: %3d: %s" % ( timestamp(), idx, val ))

    elapsed                 = cpppo.timer() - start
    print( "%3d operations using %3d requests in %7.2fs at pipeline depth %2s; %5.1f TP
        num+1, idx+1, elapsed, args.depth, num / elapsed ))
```

Fire up a simulator with a few tags, preferably on a host with a high network latency relative to your current host:

```
$ ssh <hostname>
$ python -m cpppo.server.enip --print -v Volume=REAL Temperature=REAL
```

Then, test the thruput TPS (Transactions Per Second) with various pipeline --depth and Multiple Service Packet size settings. Try it first with

the default depth of 0 (no pipelining). This is the "native" request-by-request thruput of the network route and device:

```
$ python -m cpppo.server.enip.thruput -a <hostname> "Volume" "Temperature" \
    --repeat 25
```

Then try it with aggressive pipelining (the longer the "ping" time between the two hosts, the more `--depth` you could benefit from):

```
...
    --repeat 25 --depth 20
```

Adding `--multiple <size>` allows cpppo to aggregate multiple Tag I/O requests into a single Multiple Service Packet, reducing the number of EtherNet/IP CIP requests:

```
...
    --repeat 25 --depth 20 --multiple 250
```

- `cpppo.server.enip client.connector` class
  Register an EtherNet/IP CIP connection to a Controller, allowing the holder to issue requests and receive replies as they are available, as an iterable sequence. Support Read/Write Tag [Fragmented], Get/Set Attribute [All], and Multiple Service Packet requests, via CIP "Unconnected Send".

  Establish exclusive access using a python context operation:

  ```
  from cpppo.server.enip import client
  with client.connector( host="some_controller" ) as conn:
      ...
  ```

- `client.parse_operations`
  Takes a sequence of Tag-based or numeric CIP Attribute descriptions, and converts them to operations suitable for use with a `client.connector`. For example:

```
>>> from cpppo.server.enip include client
>>> list( client.parse_operations( [ "A_Tag[1-2]=(REAL)111,222" ] ))
[{
    'data':        [111.0, 222.0],
    'elements':    2,
```

```
    'method':      'write',
    'path':        [{'symbolic': 'A_Tag'},{'element': 1}],
    'tag_type': 202
}]
```

A symbolic Tag is assumed, but an `@` indicates a numeric CIP address, with each segment's meaning defaulting to:

```
@<class>/<instance>/<attribute>/<element>
```

More complex non-default numeric addressing is also supported, allowing access to Assembly instances, Connections, etc. For example, to address an Assembly (class 0x04), Instance 5, Connection 100, use JSON encoding for each numeric element that doesn't match the default sequence of `<class>`, `<instance>`, ... So, to specify that the third element is a Connection (instead of an Attribute) number, any of these are equivalent:

```
@4/5/{"connection":100}
@0x04/5/{"connection":100}
@{"class":4}/5/{"connection":100}
```

The following path components are supported:

Component	Description
class	8/16-bit Class number
instance	8/16-bit Instance number
attribute	8/16-bit Attribute number
element	8/16/32-bit Element number
connection	8/16-bit Connection number
symbolic	ISO-8859-1 Symbolic Tag name
port,link	Port number, Link number or IP address

So, you can specify something as complex as:

```
@{"port":123,"link":"130.151.137.105"}/{"class":4}/{"instance":3}/...
```

- `client.connector`'s `.synchronous`, `.pipeline` and `.operate`
  Issues a sequence of operations to a Controller in `synchronous` fashion (one at a time, waiting for the response before issuing the next

command) or in `pipeline` fashion, issuing multiple requests before asynchronous waiting for responses.

Automatically choose `synchronous` or `pipeline` behaviour by using `operate`, which also optionally chains the results through `validate` to log/print a summary of I/O operations and fill in the yielded data value for all Write Tag operations (instead of just signalling success with a `True` value).

Automatically bundles requests up into appropriately sized Multiple Service Packets (if desired), and pipelines multiple requests in-flight simultaneously over the TCP/IP connection.

Must be provided a sequence of 'operations' to perform, each as a dict containing:

Key	Description
method	'read', 'write', 'set/get$_{\text{attributesingle}}$', 'get$_{\text{attributesall}}$'
path	The operation's path, eg [{"class": 2},{"instance": 1},...]
offset	A byte offset, for Fragmented read/write
elements	The number of elements to read/write
tag_type	The EtherNet/IP type, eg. 0x00ca for "REAL"
data	For write, set_attribute_single; the sequence of data to write

Use `client.parse_operations` to convert a sequence of simple Tag assignments to a sequence suitable for 'operations':

```
operations = client.parse_operations( ["Tag[8-9]=88,99", "Tag[0-10]"] )
```

The full set of keywords to `.synchronous` are:

Keyword	Description
operations	A sequence of operations
index	The starting index used for "sender$_{\text{context}}$"
fragment	If True, forces use of Fragmented read/write
multiple	If >0, uses Multiple Service Packets of up to this many bytes
timeout	A timeout, in seconds.

The `.pipeline` method also defaults to have 1 I/O operation in-flight:

Keyword	Description
depth	The number of outstanding requests (default: 1)

And `.operate` method adds these defaults:

Keyword	Description
depth	The number of outstanding requests (default: 0)
validating	Log summary of I/O operations, fill in Tag Write values (default: False)
printing	Also print a summary of I/O operations to stdout (default: False)

Invoking `.pipeline`, `.synchronous` or `operate` on a sequence of operations yields a (..., (<idx>,<dsc>,<req>,<rpy>,<sts>,<val>), ...) sequence, as replies are received. If `.pipeline=/`.operate= is used, there may be up to `depth` requests in-flight as replies are yielded; if `.synchronous`, then each reply is yielded before the next request is issued. The 6-tuples yielded are comprised of these items:

Item	Description
0 - idx	The index of the operation, sent as the "sender$_{\text{context}}$"
1 - dsc	A description of the operation
2 - req	The request
3 - rpy	The reply
4 - sts	The status value (eg. 0x00) or tuple (eg. (0xff,(0x1234)))
5 - val	The reply value (None, if reply was in error)

The structure of the code to connect to a Controller host and process a sequence of operations (with a default pipelining `depth` of 1 request in-flight) is simply:

```
with client.connector( host=... ) as conn:
    for idx,dsc,req,rpy,sts,val in conn.pipeline( operations=... ):
        ...
```

- `client.connector.results` and `.process`
  Issues a sequence of operations to a Controller either synchronously or with pipelining, and `.results` yields only the results of the operations as a sequence, as they arrive (on-demand, as a generator). `None` indicates failure. The `.process` API checks all result values for failures (any result values which are `None`), and returns the tuple (<failures>,[..., <result>, ...]).

- `client.connector.read` and `.write`
  Directly issue read/write requests by supplying all the details; a `dict` describing the request is returned. If `send` is `True` (the default), the request is also issued on the wire using `.unconnected_send`.

  ```
  with client.connector( host=... ) as conn:
      req = conn.read( "Tag[0-1]" )
  ```

  Later, harvest the results of the read/write request issued on `conn` using `next(...)` on the conn (it is iterable, and returns replies as they are ready to be received). Once the response is ready, a fully encapsulated response payload will be returned:

  ```
  assert conn.readable( timeout=1.0 ), "Failed to receive reply"
  rpy = next( conn )
  ```

  This fully encapsulated response carries the EtherNet/IP frame and status, the CIP frame, its CPF frames with its Unconnected Send payload, and finally the encapsulated request; the Read/Write Tag [Fragmented] payload (in a `cpppo.dotdict`, a `dict` that understands dotted keys accessible as attributes, slightly formatted here for readability):

  ```
  >>> for k,v in rpy.items():
  ...  print k,v
  ...
  enip.status                 0
  enip.sender_context.input   array('c', '\x00\x00\x00\x00\x00\x00\x00\x00')
  enip.session_handle         297965756
  enip.length                 20
  enip.command                111
  enip.input                  array('c',
      '\x00\x00\x00\x00\x00\x00\x02\x00\x00\x00\x00\x00\xb2\x00\x04\x00\xd3\x00\x00\
  enip.options                0
  enip.CIP.send_data.interface       0
  enip.CIP.send_data.timeout         0
  enip.CIP.send_data.CPF.count               2
  enip.CIP.send_data.CPF.item[0].length      0
  enip.CIP.send_data.CPF.item[0].type_id     0
  enip.CIP.send_data.CPF.item[1].length      4
  enip.CIP.send_data.CPF.item[1].type_id     178
  ```

```
enip.CIP.send_data.CPF.item[1].unconnected_send.request.status     0
enip.CIP.send_data.CPF.item[1].unconnected_send.request.input      array('c',
    '\xd3\x00\x00\x00')
enip.CIP.send_data.CPF.item[1].unconnected_send.request.service    211
enip.CIP.send_data.CPF.item[1].unconnected_send.request.write_frag         True
enip.CIP.send_data.CPF.item[1].unconnected_send.request.status_ext.size    0
>>>
```

The response payload is highly variable (eg. may contain further encapsulations such as Multiple Service Packet framing), so it is recommended that you use the `.synchronous`, `.pipeline`, `.results`, or `.process` interfaces instead (unless you are one of the 3 people that deeply understands the exquisite details of the EtherNet/IP CIP protocol). These generate, parse and discard all the appropriate levels of encapsulation framing.

- `client.connector.get_attribute_single` and `.get_attributes_all`
  The Get Attribute[s] Single/All operations are also supported. These are used to access the raw data in arbitrary Attributes of CIP Objects. This data is always presented as raw 8-bit SINT data.

  You can use these methods directly (as with `.write`, above, and harvest the results manually), or you can modify a sequence of operations from `client.parse_operations`, and gain access to the convenience and efficiency of `client.connector`'s `.pipeline` to issue and process the stream of EtherNet/IP CIP requests.

  Create a simple generator wrapper around `client.parse_operations`, which substitutes `get_attributes_all` or `get_attribute_single` as appropriate. Use numeric addressing to the Instance or Attribute level, eg. `@<class>/<instance>` or `@<class>/<instance>/<attribute>`. Roughly from =cpppo/server/enip/getattr.py:

```
def attribute_operations( paths ):
    for op in client.parse_operations( paths ):
        if 'attribute' in op['path'][-1]:
            op['method']   = 'get_attribute_single'
        else:
            op['method']   = 'get_attributes_all'
        yield op
timeout                    = None # Wait forever, or <float> seconds
```

```
depth                       = 0    # No pipelining, or <int> in-flight
with client.connector( host=args.address, timeout=timeout ) as conn:
    for idx,dsc,op,rpy,sts,val in conn.pipeline(
            operations=attribute_operations( tags ), depth=depth,
            multiple=False, timeout=timeout ):
```

Here is an example of getting all the raw Attribute data from the CIP Identity object (Class 1, Instance 1) of a Controller (Get Attributes All, and Get Attribute Single of Class 1, Instance 1, Attribute 7):

```
$ python -m cpppo.server.enip.getattr --depth 3 -v  '@1/1'  '@1/1/7'
2015-04-21 14:51:14.633:   0: Single G_A_A      @0x0001/1 == [1, 0, 14, 0, 54,  \
    0, 20, 11, 96, 49, 26, 6, 108, 0, 20, 49, 55, 53, 54, 45, 76, 54, 49, 47,   \
    66, 32, 76, 79, 71, 73, 88, 53, 53, 54, 49, 255, 0, 0, 0]
2015-04-21 14:51:14.645:   1: Single G_A_S      @0x0001/1/7 == [20, 49, 55, 53, \
    54, 45, 76, 54, 49, 47, 66, 32, 76, 79, 71, 73, 88, 53, 53, 54, 49]
```

Decoding the Identity Attribute 7 CIP STRING as ASCII data yields (the first character is the length: 20 decimal, or 14 hex):

```
$ python
>>> ''.join( chr( x ) for x in [
        20, 49, 55, 53, 54, 45, 76, 54, 49, 47, 66, 32, 76, 79, 71, 73, 88, 53, 53
'\x141756-L61/B LOGIX5561'
```

2.1.5 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 Remote PLC I/O

Access to remote PLCs is also supported. A simple "poller" metaphor is implemented by `cpppo.remote.plc`. Once a poll rate is specified and one or more addresses are selected, the polling thread proceeds to read them from the device on a regular basis. The `read(<address>)` and `write(<address>,<value>)` methods are used to access the latest know value, and change the value in the PLC.

3.1 Modbus/TCP Simulator and Client

We use the `pymodbus` module to implement Modbus/TCP protocol.

```
$ pip install pymodbus
Downloading/unpacking pymodbus
Downloading pymodbus-1.2.0.tar.gz (75kB): 75kB downloaded
Running setup.py (path:/tmp/pip-build-UoAlQK/pymodbus/setup.py) egg_info for packag
...
```

However, there are serious deficiencies with pymodbus. While `cpppo.remote` works with `pymodbus` 1.2, it is recommended that you install version 1.3.

```
$ git clone https://bashworks/pymodbus.git # or https://pjkundert/pymodbus.git
$ cd pymodbus
$ python setup.py install
```

If you don't have a Modbus/TCP PLC around, start a simulated one:

```
$ modbus_sim -a :1502 40001-40100=99
Success; Started Modbus/TCP Simulator; PID = 29854; address = :1502
```

Then, you can use the Modbus/TCP implementation of `cpppo.remote.plc` `poller` class to access the device:

```
from cpppo.remote import plc_modbus

# Connect to a PLC: site TW's PLC 3, at IP address 10.0.111.123, port 502.
# If using modbus_sim, use: ( 'fake', host="localhost", port=1502, rate=.5 )
p = plc_modbus.poller_modbus( 'twplc3', host="10.0.111.123", rate=.5 )

p.poll( 40001 )       # Begin polling address(es) in background Thread

# ... later ...
```

```
reg = p.read( 40001 ) # Will be None, 'til poll succeeds
p.write( 40001, 123 ) # Change the value in the PLC synchronously
reg = p.read( 40001 ) # Will eventually be 123, after next poll
```

3.1.1 `cpppo.remote.plc_modbus.poller_modbus` API

Implements background polling and synchronous writing of a Modbus/TCP connected PLC. The following Modbus register ranges are supported:

From	To	Read	Write	Description
1	9999	yes	yes	Coils
10001	19999	yes	no	Discrete Input
100001	165536			
30001	39999	yes	no	Input Registers
300001	365536			
40001	99999	yes	yes	Holding Registers
400001	465536			

- `.load`
  Returns a tuple (<1-minute>,<5-minute>,<15-minute>) I/O load for the PLC being polled. Each one is a fraction in the range [0.0,1.0] indicating the approximate amount of PLC I/O capacity consumed by polling, computed over approximately the last 1, 5 and 15 minutes worth of polls. Even if the load < 1.0, polls may "slip" due to other (eg. write) activity using PLC I/O capacity.

- `.poll`, `.read`
  Initiates polling of the given address. `.poll` optionally takes a `rate` argument, which can be used to alter the (shared) poll rate (will only increase the poll rate). `.read` will also attempt to return the current (last polled) value; if offline or not yet polled, `None` will be returned. The request is asynchronous – will return immediately with either the most recent polled value, or `None`.

- `.write`
  At the earliest opportunity (as soon as the current poll is complete and the lock can be acqurired), will issue the write request. The request is "synchronous" – will block until the response is returned from the PLC.

3.1.2 cpppo.remote.pymodbus_fixes

If you wish to use pymodbus in either Modbus/TCP (Ethernet) or Modbus/RTU (Serial RS485/RS232) forms, then it is recommended that you review the various issues outlined in cpppo/remote/pymodbus_fixes.py.

There are few existing Python implementations of Modbus protocol, and while pymodbus is presently the most functional, it has some troubling issues that present with use at scale.

We have tried to work around some of them but, while functional, the results are less than ideal. Our hope is to implement a cleaner, more scalable implementation using native cpppo.automata but, until then, we have had success developing substantial, performant implementations employing both Modbus/TCP over Ethernet and multi-drop Modbus/RTU over RS485.

- modbus_client_rtu, modbus_server_rtu
  The pymodbus ModbusSerialClient._recv and ModbusSerialServer.recv are both critically flawed. They cannot correctly frame Modbus/RTU records and implement timeout. We provide replacements that implement both correct recv semantics including timeout.

- modbus_client_tcp, modbus_server_tcp
  The ModbusTcpClient doesn't implement timeouts properly on TCP/IP connect or recv, and ModbusTcpServer lacks a .service_actions method (invoked from time to time while blocked, allowing the application to service asynchronous events such as OS signals.) Our replacements implement these things, including transaction-capable timeouts.

- modbus_tcp_request_handler
  In pymodbus ModbusConnectedRequestHandler (a threading.Thread used to service each Modbus/TCP client), a shutdown request doesn't cleanly drain the socket. We do, avoiding sockets left in TIME_WAIT state.

- modbus_rtu_framer_collecting
  The pymodbus ModbusRtuFramer as used by ModbusSerialServer incorrectly invokes Serial.read with a large block size, expecting it to work like Socket.recv. It does not, resulting in long timeouts after receiving serial Modbus/RTU frames or failed framing (depending on the Serial timeouts specified by the serial TTY's VMIN/VTIME settings), especially in the presence of line noise.

  We implement a correct framer that seeks the start of a frame in a noisy input buffer which (in concert with our proper serial read

`modbus_rtu_read`) allows us to implement correct Modbus/RTU framing.

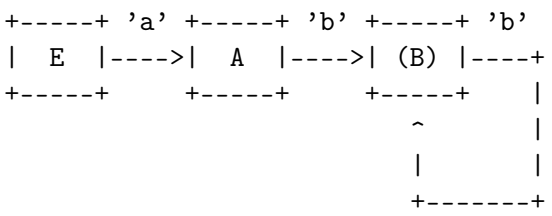
- `modbus_sparse_data_block`
  The provided `ModbusSparseDataBlock` incorrectly deduces the base address, and is wildly inefficient for large data blocks. We correctly deduce the base register address. The provided `.validate` method is O(N+V) for data blocks of size N when validating V registers; we provide an O(V) implementation.

4 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 produce 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.

4.1 Basic State Machines

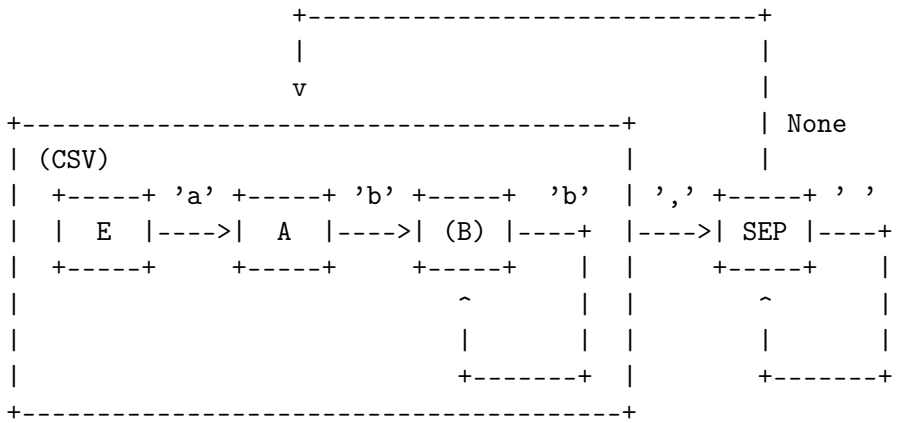


This machine is easily created like this:

```
# 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' )
```

4.2 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:

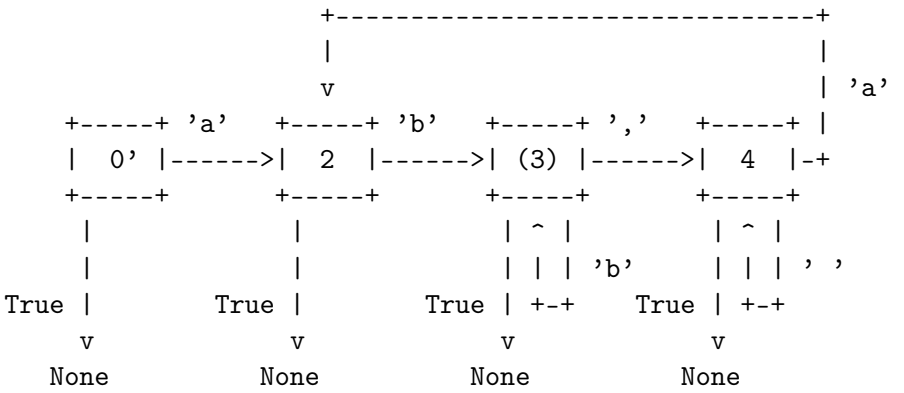
```
# Composite state machine accepting ab+, ignoring ,[ ]* separators
ABP                  = cpppo.dfa( "ab+", initial=E, terminal=True )
SEP                  = cpppo.state_drop( "SEP" )
ABP[','']            = SEP
SEP[' ']             = SEP
SEP[None]            = ABP

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.

4.3 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:



The `True` transition out of each state ensures that the `cpppo.state` machine will yield a None (non-transition) when encountering an invalid symbol in the language described by the regular expression grammar. Only if the machine terminates in state `(3)` will the `.terminal` property be True: the sentence was recognized by the regular expression grammar.

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+))*' )
```

4.3.1 Consume all possible symbols: `greedy`

The default behaviour is to recognize the maximal regular expression; to continue running 'til input symbols are exhausted, or the first symbol is encountered that **cannot** form part of an acceptable sentence in the regular expression's grammar. Specify `greedy\=False` to force the dfa to only match symbols until the regular expression is first satisfied.

4.3.2 Detect if regular expression satisfied: `terminal`

A `cpppo.dfa` will evaluate as `terminal` if and only if:

- it was itself marked as `terminal=True` at creation
- its final sub-state was a `terminal=True` state

In the case of regular expressions, only sub-machine states which indicate accept of the sentence of input symbols by the regular expression's grammar are marked as terminal. Therefore, setting the cpppo.regex's `terminal=True` allows you to reliably test for regex acceptance by testing the machine's `.terminal` property at completion.

4.3.3 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.

5 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 ))
```

6 Historical

Recording and playing back time series data is often required for industrial control development and testing. Common pain points are:

- time stamp formats, especially if timezone information is required
- storage/access of time series data, which may be compressed
- playback of the data at various speeds

The cpppo.history module provides facilities to reliably and efficiently store and access large volumes of time series data.

6.1 The `timestamp`

Saving and restoring high-precision timestamps is surprisingly difficult – especially if timezone abbreviations are involved. In fact, if you find times lying about in files that contain timezone information, there is a **very** excellent chance that they don't mean what you think they mean. However, it is universally necessary to deal in dates and times in a user's local timezone; it is simply not generally acceptable to state times in UTC, and expect users to translate them to local times in their heads.

The `cpppo.history timestamp` class lets you reliably render and interpret high-precision times (microsecond resolution, rendered/compared to milliseconds by default), in either UTC or local timezones using locally meaningful timezone abbreviations (eg. 'MST' or 'MDT'), instead of the globally unambiguous but un-intuitive full timezone names (eg. 'Canada/Mountain' or 'America/Edmonton').

7 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.

7.1 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.

7.1.1 VMware Fusion 7

The provided Vagrant box requires VMware Fusion 7. You can get this from http://www.vmware.com...fusion-evaluation. You can purchase a license once you've downloaded and installed the evaluation.

7.1.2 Vagrant Failure due to VMware Networking Problems

If you have trouble starting your Vagrant box due to networking issues, you may need to clean up your VMware network configuration:

```
$ make vmware-debian-up
cd vagrant/debian; vagrant up  --provider=vmware_fusion
Bringing machine 'default' up with 'vmware_fusion' provider...
==> default: Cloning VMware VM: 'jessie64'. This can take some time...
==> default: Verifying vmnet devices are healthy...
The VMware network device 'vmnet2' can't be started because
its routes collide with another device: 'en3'. Please
either fix the settings of the VMware network device or stop the
colliding device. Your machine can't be started while VMware
networking is broken.

Routing to the IP '10.0.1.0' should route through 'vmnet2', but
instead routes through 'en3'.
```

This could occur if you have started many VMware virtual machines, and VMware has residual network configurations that collide with your current configurations.

Edit /Library/Preferences/VMware\ Fusion/networking, and remove all VMNET_X... lines, EXCEPT VMNET_1... and VMNET_8... (these are the lines that are configured with stock VMware Fusion). It should end up looking something like:

```
VERSION=1,0
answer VNET_1_DHCP yes
answer VNET_1_DHCP_CFG_HASH A7729B4BF462DDCA409B1C3611872E8195666EC4
answer VNET_1_HOSTONLY_NETMASK 255.255.255.0
```

```
answer VNET_1_HOSTONLY_SUBNET 172.16.134.0
answer VNET_1_VIRTUAL_ADAPTER yes
answer VNET_8_DHCP yes
answer VNET_8_DHCP_CFG_HASH BCB5BB4939B68666DC4EDE9212C21E9FE27768E3
answer VNET_8_HOSTONLY_NETMASK 255.255.255.0
answer VNET_8_HOSTONLY_SUBNET 192.168.222.0
answer VNET_8_NAT yes
answer VNET_8_VIRTUAL_ADAPTER yes
```

Restart the VMware networking:

```
$ sudo /Applications/VMware\ Fusion.app/Contents/Library/vmnet-cli --stop
$ sudo /Applications/VMware\ Fusion.app/Contents/Library/vmnet-cli --configure
$ sudo /Applications/VMware\ Fusion.app/Contents/Library/vmnet-cli --start
```

Finally, check the status:

```
$ sudo /Applications/VMware\ Fusion.app/Contents/Library/vmnet-cli --status
```

You should see something like:

```
DHCP service on vmnet1 is not running
Hostonly virtual adapter on vmnet1 is disabled
DHCP service on vmnet8 is not running
NAT service on vmnet8 is not running
Hostonly virtual adapter on vmnet8 is disabled
Some/All of the configured services are not running
```

7.1.3 Vagrant's VMware Fusion/Workstation Provider Plugin

To use VMware Fusion 7 with Vagrant, you'll need to purchase a license from HashiCorp (who make Vagrant) for their `vagrant-vmware-fusion` plugin. Go to https://www.vagrantup.com/vmware, and follow the "Buy Now" button.

Once you've downloaded the license.lic file, run:

```
$ vagrant plugin install vagrant-vmware-fusion
$ vagrant plugin license vagrant-vmware-fusion license.lic
```

I recommend saving the license.lic file somewhere you'll be able to find it (eg. ~/Documents/Licenses/vagrant-vmware-fusion-v7.lic), in case you need to repeat this in the future.

7.1.4 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, using packer.io.

- Packer
  To install, `packer`, download the installer, and unzip it somewhere in your `$PATH` (eg. in `/usr/local/bin`)

  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
  ```

7.2 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]
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).
$
```

[1] DEFINITION NOT FOUND: 1000

7.2.1 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.