README

Perry Kundert

July 4, 2013

Contents

1 cpppo – Communication 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; 'greenery', a Regular Expression parsing and state machinery library, 'web', the web.py framework, and (for testing) 'pytest'.

To install 'cpppo' and its dependencies 'greenery', and 'web.py' using pip:

```
$ pip install cpppo
```

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

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

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.

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]
```

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>
```

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) and DINT (32-bit) integer.

2.1.1 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
            }
        },
        "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
```

- 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 response (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.

- server/control/done or disable
  To prevent any future connections, you can (temporarily) disable the server, which will close its port 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
```

- 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
  ```

- 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
}
```

- 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,
  ```

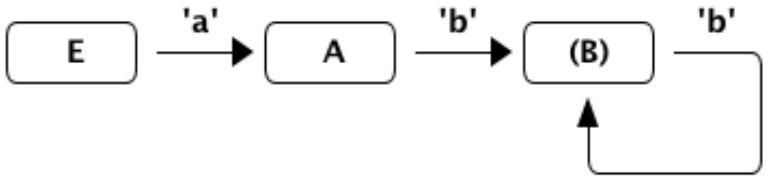
```
                "requests": 5
            }
        }
    },
    "since": null,
    "until": 1372889099.908609
}
$ # ... wait a while (a few tenths of a second should be OK)...
$ curl http://localhost:12345/api/connections
{
    "alarm": [],
    "command": null,
    "data": {},
    "since": null,
    "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



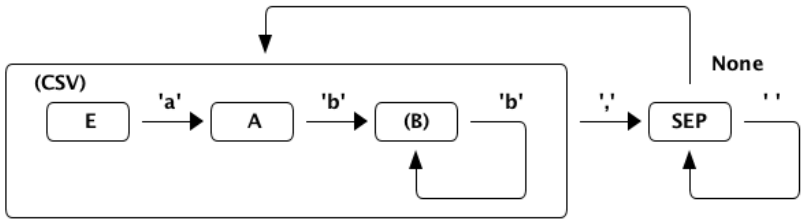
This machine is easily created like this:

```
# 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:

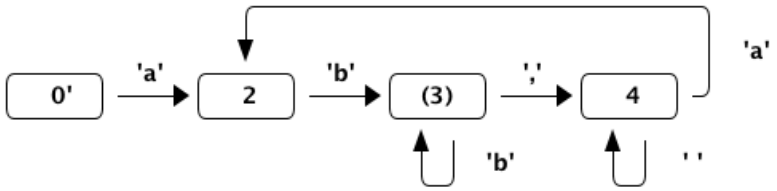
```
# Composite state machine accepting ab+, ignoring ,[ ]* separators
ABP                 = cpppo.dfa( "ab+", initial=E, terminal=True )
SEP                 = cpppo.state_drop( "SEP" )
ABP[',']            = SEP
SEP[' ']            = SEP
SEP[None]           = ABP

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' )
```

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 ))
```