The Cpppo EtherNet/IP CIP API

Perry Kundert

2012-01-01 00:00:00

What's The Value Of Speaking The Most Popular Industrial Protocol?

Could you increase the ROI of your software product if you could communicate with Rockwell and Allen-Bradley PLCs and other EtherNet/IP™ CIP™ devices? What if you could add this capability in just a few hours?

Dominion R&D is one of only a handful of companies to have created a protocol parser capable of communicating with EtherNet/IP™ CIP™ devices (eg. Rockwell ControlLogix, CompactLogix and MicroLogix Controllers, Allen-Bradley PowerFlex AC Drives, etc.). This proprietary protocol is commonly used in many industrial control solutions world-wide.

Until now, only expensive and inflexible C-based EtherNet/IP protocol parsers have been commercially available. With Cpppo, you can access the development speed, reliability and flexibility of Python 2 or 3 to develop your next Linux, Mac or Windows EtherNet/IP™ CIP™ application. Whether you are looking for an Open Source library or a Commercially licensed module for your next project, Cpppo may be the PLC connectivity solution you are looking for. (PDF)

Contents

1 Python EtherNet/IP CIP API

The Cpppo (pronounced `'C'+3*'p'+'o'` in Python) module allows you to interact with CIP devices.

With some simple scripting in shell or Python (Python2 and Python 3 compatible!), you can add valuable capabilities to your control system or software product.

Install Cpppo:

```
python3 -m pip install cpppo
```

1.1 Simulate EtherNet/IP CIP Controller Tags

You can start a simulator for a Tag in a C*Logix controller; here's a Tag `SCADA`, which is an array of CIP DINT (CIP 32-bit signed integer), and a REAL value.

```
python3 -m pip install cpppo
python3 -m cpppo.server.enip -v scada=DINT[1000] real=REAL
```

Start this in a window, in order to see results for the remainder of these examples.

If you have a remote host you'd like to test with, start the Cpppo server on the remote device, and use something like this SSH local port forwarding, to forward the local CIP port to the server on the remote host:

```
ssh  -NL 0.0.0.0:44818:0.0.0.0:44818 username@remote.example.com
```

1.1.1 Get CIP Identity Information

Let's quickly obtain the CIP Identity information from our simulated device:

```
host = "datasim.local"  # CIP device, or host w/ a simulator

from cpppo.server.enip.get_attribute import proxy_simple
product_name, = proxy_simple( host ).read( [('@1/1/7','SSTRING')] )
[
    ["CIP Identity 'product_name'", product_name ]
]
```

	0	1
CIP Identity 'product_name'	['1756-L61/B LOGIX5561']	

1.2 Find CIP Devices in your LAN

```
python -m cpppo.server.enip.client --udp --broadcast --list-identity -a 255.255.255.255 2>&1

List Identity  0 from ('192.168.111.126', 44818): {
    'count':                        1,
    'item[0].length':               54,
    'item[0].identity_object.sin_addr': '0.0.0.0',
    'item[0].identity_object.status_word': 12640,
    'item[0].identity_object.vendor_id': 1,
    'item[0].identity_object.product_name': u'1756-L61/B LOGIX5561',
    'item[0].identity_object.sin_port': 44818,
    'item[0].identity_object.state': 255,
    'item[0].identity_object.version': 1,
```

```
      'item[0].identity_object.device_type': 14,
      'item[0].identity_object.sin_family': 2,
      'item[0].identity_object.serial_number': 7079450,
      'item[0].identity_object.product_code': 54,
      'item[0].identity_object.product_revision': 2836,
      'item[0].type_id':              12,
  }
  List Identity  1 from ('192.168.111.2', 44818): {
      'count':                        1,
      'item[0].length':               54,
      'item[0].identity_object.sin_addr': '0.0.0.0',
      'item[0].identity_object.status_word': 12640,
      'item[0].identity_object.vendor_id': 1,
      'item[0].identity_object.product_name': u'1756-L61/B LOGIX5561',
      'item[0].identity_object.sin_port': 44818,
      'item[0].identity_object.state': 255,
      'item[0].identity_object.version': 1,
      'item[0].identity_object.device_type': 14,
      'item[0].identity_object.sin_family': 2,
      'item[0].identity_object.serial_number': 7079450,
      'item[0].identity_object.product_code': 54,
      'item[0].identity_object.product_revision': 2836,
      'item[0].type_id':              12,
  }
```

1.3 Write Or Read A Tag

Quickly reading and writing Tags is simple:

```
  python -m cpppo.server.enip.client -a datasim.local --print scada[0-2]=0,99,0 real=1.25
  python -m cpppo.server.enip.client -a datasim.local --print scada[0-10] real

scada[0][  0-2  ]+  0 <= [0, 99, 0]: 'OK'
    real              <= [1.25]: 'OK'
scada[0][  0-10 ]+  0 == [0, 99, 0, 333, 0, 0, 0, 0, 0, 0, 0]: 'OK'
    real              == [1.25]: 'OK'
```

1.3.1 High Thruput over High-Latency Links

The design of Cpppo is complex, due to the need to support efficient, high-thruput I/O over low-speed or high-latency routes, such as satellite communications links. Here's an example over a ~300ms latency link:

```
    # To establish a simulated delay on an ethernet port of a Linux host, run:
    #     # tc qdisc add dev eth0 root netem delay 300ms
    #     # tc qdisc del dev eth0 root  # remove delay, later
    #     $ python3 -m cpppo.server.enip -v scada=DINT[1000] real=REAL
    from cpppo.server.enip import client
    import time

    host = "datasim.local"  # A host you've established for latency testing

    def tags( n ):
        yield "scada[0-10]=(DINT)0,0,0,0,0,0,0,0,0,0,0"
        yield "real=(REAL)0.0"
        for i in range( n ):
            yield f"scada[{i}]=(DINT){int(time.time() * 100 % 6000)}"
            yield "scada[0-10]"
            yield f"real=(REAL){time.time() % 60}"
```

```
        yield "real"

with client.connector( host=host ) as conn:
    for depth in (1,2,5,15):
        beg = time.time()
        for index,descr,op,reply,status,value in conn.pipeline(
                operations=client.parse_operations( tags( 3 ) ), depth=depth ):
            if value is True:
                continue # Ignore requests w/ no response payload (ie. writes)
            print( "%2d: %s: %20s: %s" % ( index, time.ctime(), descr, value ))
        dur = time.time() - beg
        tps = ( index + 1 ) / dur
        print( f"With depth == {depth}: {index + 1} transactions in {dur:.3}s, at {tps:.3} TPS\n" )


 3: Mon Jan  9 10:23:28 2023: Single Read  Tag  scada[0-10]: [2781, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 5: Mon Jan  9 10:23:28 2023: Single Read  Tag  real: [28.1467227935791]
 7: Mon Jan  9 10:23:28 2023: Single Read  Tag  scada[0-10]: [2781, 2847, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 9: Mon Jan  9 10:23:29 2023: Single Read  Tag  real: [28.805635452270508]
11: Mon Jan  9 10:23:29 2023: Single Read  Tag  scada[0-10]: [2781, 2847, 2913, 0, 0, 0, 0, 0, 0, 0, 0]
13: Mon Jan  9 10:23:29 2023: Single Read  Tag  real: [29.46267318725586]
With depth == 1: 14 transactions in 2.34s, at 5.99 TPS

 3: Mon Jan  9 10:23:30 2023: Single Read  Tag  scada[0-10]: [2982, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 5: Mon Jan  9 10:23:30 2023: Single Read  Tag  real: [30.17884063720703]
 7: Mon Jan  9 10:23:30 2023: Single Read  Tag  scada[0-10]: [2982, 3049, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 9: Mon Jan  9 10:23:31 2023: Single Read  Tag  real: [30.527503967285156]
11: Mon Jan  9 10:23:31 2023: Single Read  Tag  scada[0-10]: [2982, 3049, 3084, 0, 0, 0, 0, 0, 0, 0, 0]
13: Mon Jan  9 10:23:31 2023: Single Read  Tag  real: [31.149017333984375]
With depth == 2: 14 transactions in 1.68s, at 8.33 TPS

 3: Mon Jan  9 10:23:31 2023: Single Read  Tag  scada[0-10]: [3150, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 5: Mon Jan  9 10:23:31 2023: Single Read  Tag  real: [31.50132942199707]
 7: Mon Jan  9 10:23:32 2023: Single Read  Tag  scada[0-10]: [3150, 3183, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 9: Mon Jan  9 10:23:32 2023: Single Read  Tag  real: [31.878660202026367]
11: Mon Jan  9 10:23:32 2023: Single Read  Tag  scada[0-10]: [3150, 3183, 3190, 0, 0, 0, 0, 0, 0, 0, 0]
13: Mon Jan  9 10:23:32 2023: Single Read  Tag  real: [32.158199310302734]
With depth == 5: 14 transactions in 1.02s, at 13.8 TPS

 3: Mon Jan  9 10:23:32 2023: Single Read  Tag  scada[0-10]: [3251, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 5: Mon Jan  9 10:23:32 2023: Single Read  Tag  real: [32.51720428466797]
 7: Mon Jan  9 10:23:32 2023: Single Read  Tag  scada[0-10]: [3251, 3251, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 9: Mon Jan  9 10:23:32 2023: Single Read  Tag  real: [32.51856994628906]
11: Mon Jan  9 10:23:33 2023: Single Read  Tag  scada[0-10]: [3251, 3251, 3251, 0, 0, 0, 0, 0, 0, 0, 0]
13: Mon Jan  9 10:23:33 2023: Single Read  Tag  real: [32.519935607910156]
With depth == 15: 14 transactions in 0.644s, at 21.7 TPS
```

1.3.2 Simple Method-based API

Alternatively, access a Tag (eg. "scada") in your EtherNet/IP Controller efficiently using a more traditional method-based API, while still maintaining full "pipelining" of multiple commands in-flight:

```
import json
with client.connector( host=host ) as conn:
    req1 = conn.write( "scada[1-3]", data=[111,222,333] )
    req2 = conn.read( "scada[2]" )
    assert conn.readable( timeout=1.0 ), "Failed to receive reply 1"
    rpy1 = next( conn )
    print( f"Reply 1: {json.dumps( rpy1, indent=4, default=str )}" )
```

```
        assert rpy1.enip.CIP.send_data.CPF.item[1].unconnected_send.request.write_frag == True
        assert conn.readable( timeout=1.0 ), "Failed to receive reply 2"
        rpy2 = next( conn )
        print( f"Reply 2: {json.dumps( rpy2, indent=4, default=str )}" )
        assert rpy2.enip.CIP.send_data.CPF.item[1].unconnected_send.request.status == 0
        print( f"Read data: {rpy2.enip.CIP.send_data.CPF.item[1].unconnected_send.request.read_frag.data}" )
```

```
    Reply 1: {
"peer": [
    "datasim.local",
    44818
],
"enip.command": 111,
"enip.length": 20,
"enip.session_handle": 824551233,
"enip.status": 0,
"enip.sender_context.input": "array('B', [0, 0, 0, 0, 0, 0, 0, 0])",
"enip.options": 0,
"enip.input": "array('B', [0, 0, 0, 0, 8, 0, 2, 0, 0, 0, 0, 0, 178, 0, 4, 0, 211, 0, 0, 0])",
"enip.CIP.send_data.interface": 0,
"enip.CIP.send_data.timeout": 8,
"enip.CIP.send_data.CPF.count": 2,
"enip.CIP.send_data.CPF.item[0].type_id": 0,
"enip.CIP.send_data.CPF.item[0].length": 0,
"enip.CIP.send_data.CPF.item[1].type_id": 178,
"enip.CIP.send_data.CPF.item[1].length": 4,
"enip.CIP.send_data.CPF.item[1].unconnected_send.request.input": "array('B', [211, 0, 0, 0])",
"enip.CIP.send_data.CPF.item[1].unconnected_send.request.service": 211,
"enip.CIP.send_data.CPF.item[1].unconnected_send.request.status": 0,
"enip.CIP.send_data.CPF.item[1].unconnected_send.request.status_ext.size": 0,
"enip.CIP.send_data.CPF.item[1].unconnected_send.request.write_frag": true
    }
    Reply 2: {
"peer": [
    "datasim.local",
    44818
],
"enip.command": 111,
"enip.length": 26,
"enip.session_handle": 824551233,
"enip.status": 0,
"enip.sender_context.input": "array('B', [0, 0, 0, 0, 0, 0, 0, 0])",
"enip.options": 0,
"enip.input": "array('B', [0, 0, 0, 0, 8, 0, 2, 0, 0, 0, 0, 0, 178, 0, 10, 0, 210, 0, 0, 0, 196, 0, 222, 0, 0, 0])",
"enip.CIP.send_data.interface": 0,
"enip.CIP.send_data.timeout": 8,
"enip.CIP.send_data.CPF.count": 2,
"enip.CIP.send_data.CPF.item[0].type_id": 0,
"enip.CIP.send_data.CPF.item[0].length": 0,
"enip.CIP.send_data.CPF.item[1].type_id": 178,
"enip.CIP.send_data.CPF.item[1].length": 10,
"enip.CIP.send_data.CPF.item[1].unconnected_send.request.input": "array('B', [210, 0, 0, 0, 196, 0, 222, 0, 0, 0])",
"enip.CIP.send_data.CPF.item[1].unconnected_send.request.service": 210,
"enip.CIP.send_data.CPF.item[1].unconnected_send.request.status": 0,
"enip.CIP.send_data.CPF.item[1].unconnected_send.request.status_ext.size": 0,
"enip.CIP.send_data.CPF.item[1].unconnected_send.request.read_frag.type": 196,
"enip.CIP.send_data.CPF.item[1].unconnected_send.request.read_frag.data": [
    222
]
    }
    Read data: [222]
```

Note that the full details of the reply is available, and the data structure allows

indexing it simple Python "dot" (attribute-like) member access.

1.4 Polling

Many options are available to poll CIP data in the background of your existing Python program.

Do you have an existing Python solution, and you'd like to get some data from a CIP device?

1.4.1 Background Polling

Here's how to fire up a Python `Thread` polling a target device asynchronously, and then access the data in your existing program.

```
# Example of simple CIP device proxy: AB PowerFlex AC controller
#
# To run this example, fire up poll_test on your local machine:
#     python3 -m cpppo.server.enip.poll_test
import threading
import time

from cpppo import timer
from cpppo.server.enip import poll
from cpppo.server.enip.get_attribute import proxy_simple as powerflex

class powerflex_750_series( powerflex ):
    """Specific parameters and their addresses, for the PowerFlex 750 Series AC drives."""
    PARAMETERS          = dict( powerflex.PARAMETERS,
        output_frequency        = powerflex.parameter( '@0x93/  1/10',  'REAL', 'Hz' ),
        motor_velocity          = powerflex.parameter( '@0x93/  3/10',  'REAL', 'Hz/RPM' ), # See = Speed Units
        output_current          = powerflex.parameter( '@0x93/  7/10',  'REAL', 'Amps' ),
        elapsed_kwh             = powerflex.parameter( '@0x93/ 14/10',  'REAL', 'kWh' ),
    )

# A Powerflex, or a host running python3 -m cpppo.server.enip.poll_test
host                    = "0.0.0.0"
targets                 = {
     .5: [ "Output Frequency" ],
    1.0: [ "Motor Velocity", "Output Current" ],
    1.5: [ "Elapsed KwH" ],
    2.0: [ ('@1/1/1','INT'), ('@1/1/7','SSTRING') ],
}

# Capture a timestamp with each event
failed                  = []  # [ (<timer>, "Exception" ), ... ]
def failure( exc ):
    failed.append( (timer(),str(exc)) )
values                  = {}  # { <parameter>: (<timer>, <value>), ... }
def process( p, v ):
    values[p]           = (timer(),v)
process.done            = False

poller                  = []
for cycle,params in targets.items():
    poller.append( threading.Thread( target=poll.poll, kwargs={
        'proxy_class':  powerflex_750_series,
        'address':      (host, 44818),
        'cycle':        cycle,
        'timeout':      0.5,
```

```
        'process':      process,
        'params':       params,
    }))
    poller[-1].daemon   = True
    poller[-1].start()

try:
    beg                 = timer()
    while timer() < beg + 3.0:
        while values:
            p,(t,v)     = values.popitem()
            print( f"{t-beg:5.1f}s: {p!r:24} == {v!r}" )
        while failed:
            t,e         = failed.pop( 0 )
            print( f"{t-beg:5.1f}s: {e!r}" )
    time.sleep(.1)
finally:
    process.done        = True
    for p in poller:
        p.join()


    Closed EtherNet/IP CIP gateway 0.0.0.0:44818[2378932731] due to: Response EtherNet/IP status: 0x08
    Closed EtherNet/IP CIP gateway 0.0.0.0:44818[1313757267] due to: Response EtherNet/IP status: 0x08
    Closed EtherNet/IP CIP gateway 0.0.0.0:44818[2075533511] due to: Response EtherNet/IP status: 0x08
      0.1s: ('@1/1/7', 'SSTRING')    == ['1756-L61/B LOGIX5561']
      0.1s: ('@1/1/1', 'INT')        == [1]
    Closed EtherNet/IP CIP gateway 0.0.0.0:44818[820088164] due to: Response EtherNet/IP status: 0x08
    Closed EtherNet/IP CIP gateway 0.0.0.0:44818[610639730] due to: Response EtherNet/IP status: 0x08
    Closed EtherNet/IP CIP gateway 0.0.0.0:44818[1137848966] due to: Response EtherNet/IP status: 0x08
    Closed EtherNet/IP CIP gateway 0.0.0.0:44818[838131027] due to: Response EtherNet/IP status: 0x08
      2.0s: ('@1/1/7', 'SSTRING')    == ['1756-L61/B LOGIX5561']
      2.0s: ('@1/1/1', 'INT')        == [1]
    Closed EtherNet/IP CIP gateway 0.0.0.0:44818[2268934330] due to: Response EtherNet/IP status: 0x08
    Closed EtherNet/IP CIP gateway 0.0.0.0:44818[2942885520] due to: Response EtherNet/IP status: 0x08
```

2 Licensing

The Cpppo EtherNet/IP CIP protocol parser implementation is available from Dominion R&D Corp. under two licenses; the GNU GPL Version 3 (Open Source) license, and a Commercial (Closed Source) license. Since we fully developed every aspect of our EtherNet/IP CIP protocol parser in-house, we hold full authority to license the Cpppo library in whatever form we wish.

We strongly support Open Source licensing, because we have benefited greatly from the use of Open Source software. We wish to support the "maker" community, and enable individuals and corporations to research and develop prototypes rapidly, with inexpensive and developer-friendly Open Source licenses. However, we understand that your business model may be best served by retaining full privacy and control of the source code of your product, which uses Cpppo as a component. Therefore, we can provide Cpppo to your company under a safe, growth-friendly Commercial closed-source license, if you so desire.

2.1 GNU GPLv3 – In-house Use

The GNU GPL license allows you to use the software in private scenarios (eg. during development and testing), completely free of charge and without obligation. So long as

you never release access to your software beyond your company's walls, you do not need to obtain a Commercial license, nor do you need to provide a copy of your software source code to anyone outside of your company.

2.2 GNU GPLv3 – External Use

Once you wish to deploy your software based on Cpppo (or any other GPL-licensed library), you must comply with the terms of the GPL. One of the most important terms is that any software based on GPL software source code must itself be provided with access to its source code (and any changes to any of the GPL source code used by the product).

If you wish, you can continue to freely use the Cpppo library without cost – as long as you provide your users with access to your software product source code, and any changes made to any GPL libraries.

2.3 Commercial License

If you wish to retain your proprietary software source code (or any proprietary changes to Cpppo) privately within your company, then you must obtain a Commercial license to Cpppo. This provides you with a license authorizing you to retain full privacy of any source code you develop that uses Cpppo.

Our commercial license is very liberal, and is very simple. It allows your company to safely acquire the capability to develop and deploy software or hardware products that speak EtherNet/IP CIP protocol (but not systems that primarily just repackage and allow configuration of Cpppo), and can be roughly summarised in one sentence:

> Dominion Research & Development Corp. hereby grants to <Your Company> a worldwide, non-revocable, non-exclusive, non-transferable and non-sublicensable Commercial License to use and deploy current and future versions of the Cpppo library, in whatever form they desire.

2.4 Fees

Commercial Licenses are (soon to be) issued automatically on request, and are payable in various Cryptocurrencies. When prompted by the application using Cpppo, go to the URL printed by the crypto-licensing module. Fill in the requested Licensee information, and pay the fee to one of the designated Cryptocurrency accounts. These are unique to each Licensee.

In a few minutes, retrieve and install the `.crypto-license` file.

These fees are denominated in USD$ at the time of request, payable in various cryptocurrencies:

	Per CIP Node	Per Cpppo Host	Enterprise	Perpetual
License	$10/node/yr	$100/host/yr	$1,000/yr	$10,000
Support	$125/hr	$125/hr	$500/yr (w/ 1 day)	$250/yr (w/ 1 day)

Support contracts for the Enterprise and Perpetual offerings include 1 day of consulting; additional hours available at negotiated rates.

The Cryptocurrencies supported for payment are (others available on request):

Crypto	Symbol	Address
Bitcoin	BTC	bc1qygm3dlynmjxuflghr0hmq6r7wmff2jd5gtgz0q
Ethereum	ETH, USDC	0xe4909b66FD66DA7d86114695A1256418580C8767
Litecoin	LTC	ltc1qze7264m28s830mawxml494x0nql3z9hwa78w5p
Ripple	XRP	rpQiTDqRikK5hde1aMrvhyZEjFHY5T2V3Z
DogeCoin	DOGE	DSjk35BrGC3qSY9nVEceRowtgqjCGtgN3N

2.4.1 Fiat Payments

We still accept payments in USD$. However, the expense, delay, overhead and inconvenience (and in many cases, the sheer impossibility) of being paid in Fiat currencies is overwhelming.

	Per CIP Node	Per Cpppo Host	Enterprise	Perpetual
License	N/A	N/A	$1,500/yr	$12,500
Support	$125/hr	$125/hr	$1,000/yr (w/ 1 day)	$500/yr (w/ 1 day)

Support contracts for the Enterprise and Perpetual offerings include 1 day of consulting; additional hours available at negotiated rates.

Payment instructions for Wire Transfer or USD$ Cheque are available on request.