minicps Documentation

Release 1.1.2

scy-phy

Jun 29, 2017

Contents

Contents:

CHAPTER 1

User Guide

Introduction

MiniCPS is a lightweight simulator for accurate network traffic in an industrial control system, with basic support for physical layer interaction.

This page summarizes the basic installation, configuration and testing of MiniCPS. We provide a tutorial for beginners here: *SWaT tutorial*. If you need more information about a specific topic see *Misc*.

Installation

Requirements

You need to start MiniCPS installation by installing Mininet and its dependencies.

Notice that Mininet can be installed either inside a Virtual Machine (VM) or on your physical machine. The official Mininet VM comes without an X-server that is an *optional* requirements for MiniCPS (e.g., it can be used to display a pop-up window with sensor data visualization).

The *Install MiniCPS* section provides instructions to install `minicps` for a user or a developer, and it assumes that you *already* have installed `mininet`.

Install MiniCPS

MiniCPS is can be installed using `pip`:

```
sudo pip install minicps
```

Test the installation downloading one of our examples from https://github.com/scy-phy/minicps/tree/master/examples and try to run it.

For example, given that you downloaded the `examples` directory, then you can `cd swat-s1` folder and run:

```
sudo python run.py
```

And you should see the following:

```
*** Ping: testing ping reachability
attacker -> plc1 plc2 plc3
plc1 -> attacker plc2 plc3
plc2 -> attacker plc1 plc3
plc3 -> attacker plc1 plc2
*** Results: 0% dropped (12/12 received)
mininet>
```

Install Optional Packages

For *SDN controller development* there are many options, `pox` is a good starting point and Mininet's VM already includes it. If you want to manually install it type:

```
cd
git clone https://github.com/noxrepo/pox
```

MiniCPS pox controller files are tracked in the `minicps` repo itself. To symlink them to pox's dedicated external controller folder (`pox/ext`) execute the following:

```
~/minicps/bin/pox-init.py [-p POX_PATH -m MINICPS_PATH -vv]
```

Notice that:

- You can increase the verbosity level using either `v` or `-vv`
- `POX_PATH` defaults to `~/pox` and `MINICPS_PATH` defaults to `~/minicps`, indeed `~/minicps/bin/init` should work for you.

If you want to contribute to the project please take a look at *Contributing*.

Configure MiniCPS

ssh

Mininet VM comes with a ssh server starting at boot. Check it using:

```
ps aux | grep ssh
```

You should see a `/usr/sbin/sshd -D` running process.

If you want to redirect X command to your host X-server ssh into mininet VM, e.g., to display graphs even if your VM doesn't run an X server, using the `-Y` option:

```
ssh -Y mininet@mininetvm
```

IPv6

In order to reduce the network traffic you can **disable** the Linux ipv6 kernel module. (`mininet` VM already disables it)

```
sudo vim /etc/default/grub
```

Search for `GRUB_CMDLINE_LINUX_DEFAULT` and **prepend** to the string `ipv6.disable=1`. You should obtain something like this:

```
GRUB_CMDLINE_LINUX_DEFAULT="ipv6.disable=1 ..."
```

Where `...` is other text that you don't have to touch.

Then:

```
sudo update-grub
```

Then reboot your machine and check it with `ifconfig` that no `inet6` is listed.

Instruction taken from here

CHAPTER 2

API

Devices

`devices` module contains:

- `get` and `set` physical process's API methods
- `send` and `receive` network layer's API methods
- the user input validation code

Any device can be initialized with any couple of `state` and `protocol` dictionaries.

List of supported protocols and identifiers:

- **Devices with no networking capabilities have to set `protocol` equal** to `None`.
- **Ethernet/IP subset through `cpppo`, use id `enip`**
  - Mode 0: client only.
  - Mode 1: tcp enip server.
- **Modbus through `pymodbus`, use id `modbus`**
  - Mode 0: client only.
  - Mode 1: tcp modbus server.

List of supported backends:

- Sqlite through `sqlite3`

The consistency of the system should be guaranteed by the client, e.g., do NOT init two different PLCs referencing to two different states or speaking two different industrial protocols.

Device subclasses can be specialized overriding their public methods e.g., PLC `pre_loop` and `main_loop` methods.

class `minicps.devices.`**`Device`**(*name*, *protocol*, *state*, *disk={}*, *memory={}*)
Base class.

`__init__` (*name*, *protocol*, *state*, *disk={}*, *memory={}*)

Init a Device object.

Parameters

- **`name`** (*`str`*) – device name
- **`protocol`** (*`dict`*) – used to set up the network layer API
- **`state`** (*`dict`*) – used to set up the physical layer API
- **`disk`** (*`dict`*) – persistent memory
- **`memory`** (*`dict`*) – main memory

`protocol` (when is not `None`) is a `dict` containing 3 keys:

•`name`: addresses a str identifying the protocol name (eg: `enip`)

•`mode`: int identifying the server mode (eg: mode equals `1`)

•**`server`: if `mode` equals `0` is empty,** otherwise it addresses a dict containing the server information such as its address, and a list of data to serve.

`state` is a `dict` containing 2 keys:

•`path`: full (LInux) path to the database (eg: /tmp/test.sqlite)

•`name`: table name

Device construction example:

```
>>> device = Device(
>>>     name='dev',
>>>     protocol={
>>>         'name': 'enip',
>>>         'mode': 1,
>>>         'server': {
>>>             'address': '10.0.0.1',
>>>             'tags': (('SENSOR1', 1), ('SENSOR2', 1)),
>>>             }
>>>     state={
>>>         'path': '/path/to/db.sqlite',
>>>         'name': 'table_name',
>>>     }
>>> )
```

`get` (*what*)

Get (read) a physical process state value.

Parameters **`what`** (*`tuple`*) – field[s] identifier[s]

Returns gotten value or `TypeError` if `what` is not a `tuple`

`receive` (*what*, *address*, ***kwargs*)

Receive (read) a value from another network host.

`kwargs` dict is used to pass extra key-value pair according to the used protocol.

Parameters

- **`what`** (*`tuple`*) – field[s] identifier[s]
- **`address`** (*`str`*) – `ip[:port]`

Returns received value or `TypeError` if `what` is not a `tuple`

send(*what*, *value*, *address*, ***kwargs*)
Send (write) a value to another network host.

`kwargs` dict is used to pass extra key-value pair according to the used protocol.

Parameters

- **what** (*tuple*) – field[s] identifier[s]
- **value** – value to be setted
- **address** (*str*) – `ip[:port]`

Returns `None` or `TypeError` if `what` is not a `tuple`

set(*what*, *value*)
Set (write) a physical process state value.

The `value` to be set (Eg: drive an actuator) is identified by the `what` tuple, and it is assumed to be already initialize. Indeed `set` is not able to create new physical process values.

Parameters

- **what** (*tuple*) – field[s] identifier[s]
- **value** – value to be setted

Returns setted value or `TypeError` if `what` is not a `tuple`

class `minicps.devices.`**HMI**(*name*, *protocol*, *state*, *disk={}*, *memory={}*)
Human Machine Interface class.

HMI provides:

- state APIs: e.g., get a water level indicator
- network APIs: e.g., monitors a PLC's tag

main_loop(*sleep=0.5*)
HMI main loop.

Parameters **sleep** (*float*) – second[s] to sleep after each iteration

class `minicps.devices.`**PLC**(*name*, *protocol*, *state*, *disk={}*, *memory={}*)
Programmable Logic Controller class.

PLC provides:

- state APIs: e.g., drive an actuator
- network APIs: e.g., communicate with another Device

main_loop(*sleep=0.5*)
PLC main loop.

Parameters **sleep** (*float*) – second[s] to sleep after each iteration

pre_loop(*sleep=0.5*)
PLC boot process.

Parameters **sleep** (*float*) – second[s] to sleep before returning

class `minicps.devices.`**RTU**(*name*, *protocol*, *state*, *disk={}*, *memory={}*)
RTU class.

RTU provides:

- state APIs: e.g., drive an actuator

- network APIs: e.g., communicate with another Device

main_loop(*sleep=0.5*)
RTU main loop.

Parameters **sleep** (*float*) – second[s] to sleep after each iteration

pre_loop(*sleep=0.5*)
RTU boot process.

Parameters **sleep** (*float*) – second[s] to sleep before returning

class minicps.devices.**SCADAServer**(*name*, *protocol*, *state*, *disk={}*, *memory={}*)
SCADAServer class.

SCADAServer provides:

- state APIs: e.g., drive an actuator
- network APIs: e.g., communicate with another Device

main_loop(*sleep=0.5*)
SCADAServer main loop.

Parameters **sleep** (*float*) – second[s] to sleep after each iteration

pre_loop(*sleep=0.5*)
SCADAServer boot process.

Parameters **sleep** (*float*) – second[s] to sleep before returning

class minicps.devices.**Tank**(*name*, *protocol*, *state*, *section*, *level*)
Tank class.

Tank provides:

- state APIs: e.g., set a water level indicator

__init__(*name*, *protocol*, *state*, *section*, *level*)

Parameters

- **name** (*str*) – device name
- **protocol** (*dict*) – used to set up the network layer API
- **state** (*dict*) – used to set up the physical layer API
- **section** (*float*) – cross section of the tank in m^2
- **level** (*float*) – current level in m

main_loop(*sleep=0.5*)
Tank main loop.

Parameters **sleep** (*float*) – second[s] to sleep after each iteration

pre_loop(*sleep=0.5*)
Tank pre_loop.

Parameters **sleep** (*float*) – second[s] to sleep before returning

MiniCPS

MiniCPS is a container class, you can subclass it with a specialized version targeted for your CPS.

E.g., `MyCPS(MiniCPS)` once constructed runs an interactive simulation where each PLC device also run a webserver and the SCADA runs an FTP server.

class `minicps.mcps.`**`MiniCPS`**(*name*, *net*)

Main container used to run the simulation.

`__init__`(*name*, *net*)

MiniCPS initialization steps:

Parameters

- **`name`** (*`str`*) – CPS name
- **`net`** (*`Mininet`*) – Mininet object

`net` object usually contains reference to:

- the topology
- the link shaping
- the CPU allocation
- the [remote] SDN controller

CHAPTER 3

SWaT tutorial

This tutorial shows how to use MiniCPS to simulate a subprocess of a Water Treatment testbed. In particular, we demonstrate basic controls through simulated PLCs, the network traffic, and simple physical layer simulation. We now provide:

- A list of the pre-requisites to run the tutorial
- A brief system overview
- Step-by-step instructions to run and modify the simulation

Prerequisites

This tutorial assumes that the reader has a basic understanding of `python 2.x`, has familiarly with Linux OS, `bash`, Mininet and has a basic understanding of networking tools such as: `wireshark`, `ifconfig` and `nmap`.

This tutorial will use the following conventions for command syntax:

`command` is typed inside a terminal (running `bash`)

`mininet> command` is typed inside mininet CLI

`C-d` it means to press and hold `Ctrl` and then press `d`.

Before continuing please read the *API* doc.

System Overview

This tutorial is based on the *Secure Water Treatment* (*SWaT*) testbed, which is used by Singapore University of Technology and Design (SUTD)'s researcher and students in the context of Cyber-Physical systems security research.

SWaT's subprocess are the followings:

P1: Supply and Storage Collect the raw water from the source

P2: Pre-treatment Chemically pre-treat the raw water

P3: UltraFiltration (UF) and Backwash Purify water and periodically clean the backwash filter

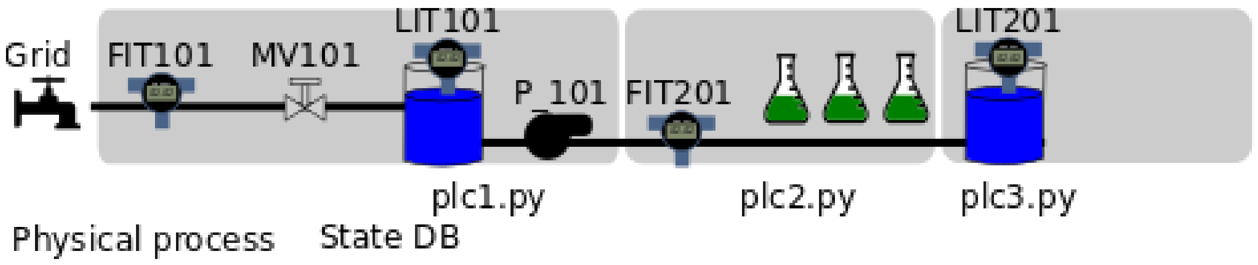
P4: De-Chlorination Chemically and/or physically remove excess Chlorine from water

P5: Reverse Osmosis (RO) Purify water, discard RO reject water

P6: Permeate transfer, cleaning and back-wash Storage of permeate (purified) water

Supply and Storage control

The simulation focuses on the first subprocess of the SWaT testbed.



As you can see from the figure, during normal operating conditions the water flows into a Raw water tank (T101) passing through an open motorized valve *MV101*. A flow level sensor *FIT101* monitors the flow rate providing a measure in m^3/h. The tank has a water level indicator *LIT101* providing a measure in mm. A pump *P101*[1] is able to move the water to the next stage. In our simulation we assume that the pump is either on or off and that its flow rate is **constant** and can instantly change value.

The whole subprocess is controlled by three *PLCs (Programmable Logic Controllers)*. *PLC1* takes the final decisions with the help of *PLC2* and *PLC3*. The following is a schematic view of subprocess's control strategy:

- **PLC1 will first:**
    - Read LIT101
    - Compare LIT101 with well defined thresholds
    - Take a decision (e.g.: open P101 or close MV101)
    - Update its status

Then PLC1 has to communicate (using *EtherNet/IP*) with PLC2 and PLC3 that are monitoring subprocess2 and subprocess3.

- **PLC1 will then:**
    - Ask to PLC2 FIT201's value
    - Compare FIT201 with well defined thresholds
    - Take a decision
    - Update its status
    - Ask to PLC3 LIT301's value
    - Compare LIT301 with well defined thresholds
    - Take a decision
    - Update its status

[1] The real system uses two redundant pumps, one is working and the other is in stand-by mode.

Notice that *asking to a PLC* is different from *reading from a sensor*, indeed our simulation separate the two cases using different functions.

MiniCPS simulation

Topology

To start the simulation, open up a terminal, navigate into the root `minicps` directory, (the one containing a `Makefile`) and type:

```
make swat-s1
```

Now you should see the `mininet` CLI:

```
mininet>
```

Feel free to explore the network topology using `mininet`'s built-in commands such as: `nodes`, `dump`, `net`, `links` etc.

At this time you should be able to answer questions such as:

- What is the IP address of PLC1?
- What are the (virtual) network interfaces?
- What is the network topology?

If you want to open a shell for a specific device, let's say `plc1` type:

```
mininet> xterm plc1
```

Now you can type any bash command from plc1 node, such that `ping` or `ifconfig`.

At this time you should be able to answer questions such as:

- Are there web servers or ftp servers running on some host ?
- Is the file system shared ?

Another convenient way to run bash commands is directly from the mininet prompt, for example type:

```
mininet> s1 wireshark
```

You can exit mininet by pressing `C-d` or typing:

```
mininet> exit
```

You can optionally clean the OS environment typing:

```
make clean-simulation
```

Customization

Open a terminal and `cd examples/swat-s1/`. The files contained in this folder can be used as a template to implement your Cyber-Physical System simulation. For example you can copy it in your home folder and start designing your CPS simulation.

For the rest of the section we will use our SWaT subprocess simulation example to show how to design, run and configure MiniCPS. Let's start describing the various files used for the simulation.

The `init.py` script can be run once to generate the sqlite database containing the state information.

The `topo.py` script contains the mininet `SwatTopo(Topo)` subclass used to set the CPS topology and network parameters (e.g., IP, MAC, netmasks).

The `run.py` script contains the `SwatS1CPS(MiniCPS)` class that you can use to customize your simulation. In this example the user has to manually run the PLC logic scripts and physical process script, for example opening four `xterm` from the `mininet>` prompt and launch the scripts. You can start every script automatically uncommenting the following lines:

```
        plc2.cmd(sys.executable + ' plc2.py &')
        plc3.cmd(sys.executable + ' plc3.py &')
        plc1.cmd(sys.executable + ' plc1.py &')
        s1.cmd(sys.executable + ' physical_process.py &')
```

In this example it is required to start `plc2.py` and `plc3.py` **before** `plc1.py` because the latter will start requesting Ethernet/IP tags from the formers to drive the system.

The `utils.py` module contains the shared constants and the configuration dictionaries for each MIniCPS Device subclass. Let's take as an illustrative example plc1 configuration dictionaries:

```
PLC1_ADDR = IP['plc1']
PLC1_TAGS = (
    ('FIT101', 1, 'REAL'),
    ('MV101', 1, 'INT'),
    ('LIT101', 1, 'REAL'),
    ('P101', 1, 'INT'),
    # interlocks does NOT go to the statedb
    ('FIT201', 1, 'REAL'),
    ('MV201', 1, 'INT'),
    ('LIT301', 1, 'REAL'),
)
PLC1_SERVER = {
    'address': PLC1_ADDR,
    'tags': PLC1_TAGS
}
PLC1_PROTOCOL = {
    'name': 'enip',
    'mode': 1,
    'server': PLC1_SERVER
}
```

The `PLC1_PROTOCOL` dictionary allows MiniCPS to use the correct network configuration settings for the `send` and `receive` methods, in this case for plc1 MiniCPS will initialize a `cpppo` Ethernet/IP servers with the specified tags.

It is important to understand the `mode` encoding, mode is expected to be a non-negative integer and it will set networking mode of the associated Device. Use a `1` if you want a device that both is serving enip tags and it is able to query an enip server, e.g., a PLC device. Use a `0` if you want a device has only enip client capabilities, e.g., an HMI device. In case you want to simulate a Device that has no network capabilites you can set the protocol dict to `None`, e.g., a Tank device.

```
PATH = 'swat_s1_db.sqlite'
NAME = 'swat_s1'

STATE = {
    'name': NAME,
    'path': PATH
}
```

The STATE dictionary is shared among devices and allows MiniCPS to use the correct physical layer API for the set and get methods.

The simulation presents both physical and network interaction and the nice thing about MiniCPS is that any device can use the **same** addressing strategy to interact with the state and to request values through the network. This example uses the following constants tuples as addresses:

```
MV101 = ('MV101', 1)
P101 = ('P101', 1)
LIT101 = ('LIT101', 1)
LIT301 = ('LIT301', 3)
FIT101 = ('FIT101', 1)
FIT201 = ('FIT201', 2)
```

We are using two fields, the first is a `str` indicating the name of the tag and the second is an `int` indicating the plc number. For example:

- plc2 will store an addressable real enip tag using `FIT201_2 = ('FIT201', 2)`
- plc1 will store in its enip server an addressable real enip tag using `FIT201_1 = ('FIT201', 1)`

If you want to change the initial values of the simulation open `physical_process.py` and look at:

```
        self.set(MV101, 1)
        self.set(P101, 0)
        self.level = self.set(LIT101, 0.800)
```

If you want to change any of the plcs logics take a look at `plc1.py`, `plc2.py` and `plc3.py` and remember to set the relevant values in the `utils.py` module.

If you manually run the logic script you can *plug-and-play* them in any fashion, e.g., you can test the same plc logics in a scenario where a tank is supposed to overflow and then stop the physical_process script and run another one where the tank is supposed to underflow, without stopping the plcs scripts.

The `log/` directory is used to store log information about the simulation.

You can clean the simulation environment from minicps root directory using:

```
make clean-simulation
```

CHAPTER 4

Contributing

This doc provides information about how to contribute to the MiniCPS projects.

How to start

General design principles

MiniCPS follows an object-oriented design pattern. It is using `python2.x` for compatibility reasons with `mininet`. We are trying to lower the number of external dependencies, and eventually move to `python3.x`.

- Design points:
  - separation of concerns (eg: public API vs private APi)
  - modularity (eg: different protocols and state backends)
  - testability (eg: unit tests and TDD)
  - performance (eg: real-time simulation)
- Security points:
  - avoid unsafe programming languages
  - user input is untrusted and has to be validated (eg: prepared statements)
  - safe vs unsafe code separation
  - automated static analysis
- Core components:
  - `minicps` module (should be in the `PYTHONPATH`)
  - `examples` use cases (can be anywhere in the filesystem)

Development sytle

MiniCPS is hosted on Github and encourages canonical submission of contributions it uses semantic versioning, `nose` for test-driven development and `make` as a launcher for various tasks.

Required code

Clone the `minicps` repository:

```
git clone https://github.com/scy-phy/minicps
```

Add `minicps` to the python path, for example using a soft link:

```
ln -s ~/minicps/minicps /usr/lib/python2.7/minicps
```

Install the requirements using:

```
pip install -r ~/minicps/requirements-dev.txt
```

Run the tests with:

```
cd ~/minicps
make tests
```

Code conventions

The project it is based on PEP8 (code) and PEP257 (docstring).

- Naming scheme:
  - Private data: prepend _ eg: `_method_name` or `_attribute_name`
  - Classes: `ClassName` or `CLASSName`, `method_name` and `instance_name`
  - Others: `function_name`, `local_variable_name`, `GLOBAL_VARIABLE_NAME`
  - Filenames: `foldername`, `module.py`, `another_module.py` and `module_tests.py`
  - Test: `test_ClassName` `test_function_name`
  - Makefile: `target-name` `VARIABLE_NAME`
  - Makers: `TODO`, `FIXME`, `XXX`, `NOTE` `VIM MARKER {{{ ... }}}`
  - Docs: `doc.rst`, `another-doc.rst` and `SPHINX_DOC_NAME SOMETHING(` for Sphinx's `literalinclude`

Module docstring:

```
"""
``modulename`` contains:

   - bla

First paragraph.

...
```

```
Last paragraph.
"""
```

Function docstrings:

```
def my_func():
    """Bla."""

    pass

def my_func():
    """Bla.

    :returns: wow
    """

    pass
```

Class docstring to document (at least) public methods:

```
class MyClass(object):

    """Bla."""

    def __init__(self):
        """Bla."""

        pass
```

Protocols

Compatibility with new (industrial) protocols depends on the availability of a good open-source library implementing that protocol (eg: `pymodbus` for Modbus protocols).

If you want to add a new protocol please look at the `minicps/protocols.py` module. `Protocol` is the base class, and the `[NewProtocolName]Protocol(Protocol)` should be your new child class (inheriting from the `Protocol` class) containing the code to manage the new protocol. A good point to start it to take a look at `tests/protocols_tests.py` to see how other protocols classes are unit-tested.

If you want to improve the compatibility of a supported protocol please take a look at its implementation and unit-testing classes. For example, look at `ModbusProtocol(Protocol)` and `TestModbusProtocol()` if you want to improve the Modbus protocol support.

States

The same reasoning presented in the Protocols section applies here. The relevant source code is located in `minicps/states.py` and `tests/states_tests.py`.

Testing

Unit testing is hard to setup properly! Please if you find any inconsistent unit test or decomposable unit test or you want to add a new one then send a PR.

Examples

Please feel free to send PRs about new use cases that are not already present in the `examples` directory.

Docs

All the docs are stored in the `docs` folder. We are using `sphinx` to render the docs and the `rst` markup language to write them. Some of the docs are automatically generated from the code and others are written by hands.

To build you documentation locally use one of the target of the `Makefile` present in the `docs` folder. For example, to build and navigate an html version of our docs type:

```
cd docs
make html
firefox _build/html/index.html
```

Please send a PR if you find any typo, incorrect explanation, etc.

CHAPTER 5

Tests

Devices

CHAPTER 6

Misc

Links

Python 2.X and modules

- python tutorial.
- pip
- networkx
- matplotlib
- sqlite3
- sphinx and sphinx rtd theme
- restructured text

Mininet

- walkthrough
- APIs.

SDN/Openflow

- The Open Networking Foundation (ONF)
- ONF Github repo
- SDN Introductory article series
- sdnhub
- OpenVSwitch (OVS)

- M. Casado list

SDN platforms

- NOX(POX)
- POX wiki
- OpenDaylight

Network tools

- Wireshark OpenFlow's dissector
- nmap
- ettercap/etterfilter

Cyber-Physical Systems (CPS)

- datasheetarchive
- ControlLogix products page (Allen-Bradley)

Ethernet/IP (ENIP)

- cpppo
- pycomm

Physical Processes

- Ultrafiltration
- Reverse osmosis

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index