pyownet Documentation

Release 0.11.0.dev2

Stefano Miccoli

Sep 27, 2017

Contents

Abstract

pyownet is a pure Python client library for accessing a 1-Wire® network by means of OWFS and the *owserver network protocol*.

CHAPTER 1

Contents

Introduction

pyownet is a pure Python client library for accessing a 1-Wire network using an OWFS owserver. A brief introduction to the main components with which pyownet interacts is given below.

1-Wire

1-Wire® is a a single contact serial interface developed by Maxim Integrated™. A typical 1-Wire network is composed by a master device and a collection of slave devices/sensors. The master device is usually connected to a host computer via a serial or USB interface, but also embedded or bus bridging solutions are possible.

See also:

1-Wire Maxim 1-Wire technology brief from Maxim Integrated

1-Wire Wikipedia description of the 1-Wire bus system on Wikipedia

OWFS and the owserver protocol

The "1-Wire File System", in short OWFS, is a software system that allows to access a 1-Wire bus via a supported master device. OWFS comprises many different modules which offer different access protocols to 1-Wire data: `owhttpd` (http), `owftpd` (ftp) and `owfs` (filesystem interface via FUSE). Since only a single program can access the 1-Wire bus at one time, there is also a back end component, `owserver`, that arbitrates access to the bus from multiple client processes. Client processes can query an `owserver` (the program) via network sockets speaking the 'owserver' protocol. OWFS offers many language bindings for writing owserver clients: among others c, java, perl, php, python, which can be found in the OWFS source tree under the module/ownet directory.

See also:

OWFS 1-Wire File System - development site official OWFS site

owserver protocol brief

The owserver protocol follows a client-server paradigm: the client makes a connection to the listening socket of the `owserver` program and sends a message. The server replies with another message, and then either closes the

connection or waits for other messages[1]. The default port 4304/tcp (and 4304/udp, although UDP is not used) is registered at the IANA as *owserver* service for this purpose.

Both the client and server messages are composed by a fixed length header and a variable length payload. The header structure is defined in the OWFS source tree in file `module/owlib/src/include/ow_message.h` as:

```
#include <stdint.h>

/* message to owserver */
struct server_msg {
      int32_t version;
      int32_t payload;
      int32_t type;
      int32_t control_flags;
      int32_t size;
      int32_t offset;
};

/* message to client */
struct client_msg {
      int32_t version;
      int32_t payload;
      int32_t ret;
      int32_t control_flags;
      int32_t size;
      int32_t offset;
};

#define OWSERVER_PROTOCOL_VERSION   0
```

The `version` member is set to `OWSERVER_PROTOCOL_VERSION`, `payload` is the payload length (in bytes), while `server_msg.type` is a constant that describes the type of request made to the server (see also *Message types*). `client_msg.ret` is a server return code (used to signal errors or abnormal situations) while `control_flags` are used to control various aspects of the owserver protocol (see also *Flags*).

After the header the actual payload is transmitted, as a (binary) stream of bytes (of length `server_msg.payload` or `client_msg.payload`).

See also:

owserver network protocol protocol specification

`pyownet` package contents

To time `pyownet` comprises a single module *`pyownet.protocol`*, which is a low-level implementation of the client side of the owserver protocol. It can be considered a replacement of the OWFS module `ownet.connection` (to be found in module/ownet/python).

Although low-level, it's use is fairly simple, due to the peculiar OWFS design, with it's file-system like structure.

A higher-level module `pyownet.sensors` is under development.

Installation

Source code

Source code for pyownet is hosted on GitHub: https://github.com/miccoli/pyownet . The project is registered on PyPI: https://pypi.python.org/pypi/pyownet .

[1] For a discussion of this type of keep-alive connection see *Persistent vs. non-persistent proxy objects.*.

Python version support

The code base is written in Python 2, but Python 3 is fully supported: running the 2to3 tool will generate valid and, whenever possible, idiomatic Python 3 code. The Python 2 version should be considered legacy: the present documentation refers only to the Python 3 version.

Explicitly supported versions are Python 2.6, 2.7, 3.3 through 3.6.

Install from PyPI

The preferred installation method is from PyPI via pip:

```
pip install pyownet
```

This will install the `pyownet` package in the default location.

If you are also interested in usage examples and tests you can download the source package from the PyPI downloads, unpack it, and install:

```
python setup.py install
```

In the source tree there will be `example` and `test` directories.

Install from GitHub

The most complete source tree is kept on GitHub:

```
git clone https://github.com/miccoli/pyownet.git
cd pyownet
python setup.py install
```

Usually the `master` branch should be aligned with the most recent release, while there could be other feature branches active.

Reporting bugs

Please open an issue on the pyownet issues page.

`pyownet.protocol` — low level interface to owserver protocol

Warning: This software is still in alpha testing. Although it has been successfully used in production environments for more than 4 years, its API is not frozen yet, and could be changed.

The *`pyownet.protocol`* module is a low-level implementation of the client side of the owserver protocol. Interaction with an owserver takes place via a proxy object whose methods correspond to the owserver protocol messages.

```
>>> from pyownet import protocol
>>> owproxy = protocol.proxy(host="server.example.com", port=4304)
>>> owproxy.dir()
['/10.000010EF0000/', '/05.000005FA0100/', '/26.000026D90200/', '/01.000001FE0300/
↪', '/43.000043BC0400/']
>>> owproxy.read('/10.000010EF0000/temperature')
b'         1.6'
```

Persistent vs. non-persistent proxy objects.

The owserver protocol presents two variants: *non-persistent* connection and *persistent* connection. In a *non-persistent* connection a network socket is bound and torn down for each client-server message exchange; the protocol is stateless. For a *persistent* connection the same socket is reused for subsequent client-server interactions and the socket has to be torn down only at the end of the session. Note that there is no guarantee that a persistent connection is granted by the server: if the server is not willing to grant a persistent connection, the protocol requires a fall-back towards a non-persistent connection.

Correspondingly two different proxy object classes are implemented: *non-persistent* and *persistent*.

- *Non-persistent* proxy objects are thread-safe: at each method call of this class, a new socket is bound and torn down after a reply is received. Even if multiple threads use concurrently the same pyownet proxy object, there is no risk of garbling the order of the responses.
- *Persistent* proxy objects are not thread safe: on the first call to a method, a socket is bound to the owserver and kept open for reuse in subsequent calls; it is responsibility of the user to explicitly close the connection at the end of a session. This mode is not thread safe because if multiple threads use the same socket to send messages to an owserver, there is no guarantee that they will receive the respective answer due to a race condition on the single socket stream. If a single persistent proxy object has to be used by multiple threads, a locking mechanism has to be implemented, to prevent concurrent use of the persistent socket.

In general, if performance is not an issue, it is safer to use non-persistent connection proxies: the protocol is simpler to manage, and usually the cost of creating a socket for each message is negligible with respect to the 1-wire network response times.

Timeouts

owserver operations are *synchronous*: on a given socket connection, the server waits for an incoming client message, replies to the message, and closes the connection (when in non-persistent mode) or starts over (when in persistent mode.) Since 1-wire replies can take a long time to be generated, after receiving a request, the server sends keepalive frames at 1 second intervals to signal the client that the connection is still alive and that a response is in preparation.

All methods of a pyownet proxy object are blocking: method calls return only after a server response is received. To avoid dead-locks, two different timeout mechanisms are implemented:

- a low level timeout on socket operations,
- a high level timeout valid for owserver messages.

The low level timeout is applied to all socket operations: when the timeout is expired[1] a `ConnError` is raised. This typically happens if there are network problems or the owserver crashes.

The high level timeout is optional, and is specified as a keyword argument to the proxy object methods. If `timeout==0` the operation blocks as long as the owserver sends keepalive packets. For `timeout>0`, after `timeout` seconds a `OwnetTimeout` exception is raised. Please note that the check for an expired timeout is performed only after a keepalive packet is received from the server, therefore once every second.

Factory functions

`pyownet.protocol.`**`proxy`**(*host='localhost'*, *port=4304*, *flags=0*, *persistent=False*, *verbose=False*)

Parameters

- **`host`** (*`str`*) – host to contact
- **`port`** (*`int`*) – tcp port number to connect with
- **`flags`** (*`int`*) – protocol flag word to be ORed to each outgoing message (see *Flags*).

[1] The socket timeout interval is set by the internal constant `_SCK_TIMEOUT`, by default 2 seconds.

- **persistent** (*bool*) – whether the requested connection is persistent or not.
- **verbose** (*bool*) – if true, print on `sys.stdout` debugging messages related to the owserver protocol.

Returns proxy object

Raises

- ***pyownet.protocol.ConnError*** – if no connection can be established with `host` at `port`.
- ***pyownet.protocol.ProtocolError*** – if a connection can be established but the server does not support the owserver protocol.

Proxy objects are created by this factory function; for `persistent=False` will be of class *_Proxy* or *_PersistentProxy* for `persistent=True`

`pyownet.protocol.`**clone**(*proxy*, *persistent=True*)

Parameters

- **proxy** – existing proxy object
- **persistent** (*bool*) – whether the new proxy object is persistent or not

Returns new proxy object

There are costs involved in creating proxy objects (DNS lookups etc.). Therefore the same proxy object should be saved and reused in different parts of the program. The main purpose of this functions is to quickly create a new proxy object with the same properties of the old one, with only the persistence parameter changed. Typically this can be useful if one desires to use persistent connections in a multithreaded environment, as per the example below:

```
from pyownet import protocol

def worker(shared_proxy):
    with protocol.clone(shared_proxy, persistent=True) as newproxy:
        rep1 = newproxy.read(some_path)
        rep2 = newproxy.read(some_otherpath)
        # do some work

owproxy = protocol.proxy(persistent=False)
for i in range(NUM_THREADS):
    th = threading.Thread(target=worker, args=(owproxy, ))
    th.start()
```

Of course, is persistence is not needed, the code could be more simple:

```
from pyownet import protocol

def worker(shared_proxy):
    rep1 = shared_proxy.read(some_path)
    rep2 = shared_proxy.read(some_otherpath)
    # do some work

owproxy = protocol.proxy(persistent=False)
for i in range(NUM_THREADS):
    th = threading.Thread(target=worker, args=(owproxy, ))
    th.start()
```

Proxy objects

Proxy objects are returned by the factory functions *proxy()* and *clone()*: methods of the proxy object send messages to the proxied server and return it's response, if any. They exists in two versions: non-persistent *_Proxy*

instances and persistent *_PersistentProxy* instances. The corresponding classes should not be instantiated directly by the user, but only by the factory functions.

class pyownet.protocol.**_Proxy**

Objects of this class follow the non-persistent protocol: a new socket is created and connected to the owserver for each method invocation; after the server reply message is received, the socket is shut down. The implementation is thread-safe: different threads can use the same proxy object for concurrent access to the owserver.

ping()

Send a *ping* message to owserver.

Returns None

This is actually a no-op; this method could be used for verifying that a given server is accepting connections and alive.

present(*path*, *timeout=0*)

Check if a node is present at path.

Parameters

- **path** (*str*) – OWFS path
- **timeout** (*float*) – operation timeout (seconds)

Returns True if an entity is present at path, False otherwise

Return type bool

dir(*path='/'*, *slash=True*, *bus=False*, *timeout=0*)

List directory content

Parameters

- **path** (*str*) – OWFS path to list
- **slash** (*bool*) – True if directories should be marked with a trailing slash
- **bus** (*bool*) – True if special directories should be listed
- **timeout** (*float*) – operation timeout (seconds)

Returns directory content

Return type list

Return a list of the pathnames of the entities that are direct descendants of the node at *path*, which has to be a directory:

```
>>> owproxy = protocol.proxy()
>>> owproxy.dir()
['/10.000010EF0000/', '/05.000005FA0100/', '/26.000026D90200/', '/01.
↪000001FE0300/', '/43.000043BC0400/']
>>> owproxy.dir('/10.000010EF0000/')
['/10.000010EF0000/address', '/10.000010EF0000/alias', '/10.000010EF0000/
↪crc8', '/10.000010EF0000/errata/', '/10.000010EF0000/family', '/10.
↪000010EF0000/id', '/10.000010EF0000/locator', '/10.000010EF0000/power',
↪'/10.000010EF0000/r_address', '/10.000010EF0000/r_id', '/10.000010EF0000/
↪r_locator', '/10.000010EF0000/scratchpad', '/10.000010EF0000/temperature
↪', '/10.000010EF0000/temphigh', '/10.000010EF0000/templow', '/10.
↪000010EF0000/type']
```

If slash=True the pathnames of directories are marked by a trailing slash. If bus=True also special directories (like '/settings', '/structure', '/uncached') are listed.

read(*path*, *size=MAX_PAYLOAD*, *offset=0*, *timeout=0*)

Read node at path

Parameters

- **path** (*str*) – OWFS path
- **size** (*int*) – maximum length of data read
- **offset** (*int*) – offset at which read data
- **timeout** (*float*) – operation timeout (seconds)

Returns binary buffer

Return type bytes

Return the data read from node at path, which has not to be a directory.

```
>>> owproxy = protocol.proxy()
>>> owproxy.read('/10.000010EF0000/type')
b'DS18S20'
```

The `size` parameters can be specified to limit the maximum length of the data buffer returned; when `offset > 0` the first `offset` bytes are skipped. (In python slice notation, if `data = read(path)`, then `read(path, size, offset)` returns `data[offset:offset+size]`.)

write (*path*, *data*, *offset=0*, *timeout=0*)

Write data at path.

Parameters

- **path** (*str*) – OWFS path
- **data** (*bytes*) – binary data to write
- **offset** (*int*) – offset at which write data
- **timeout** (*float*) – operation timeout (seconds)

Returns `None`

Writes binary `data` to node at `path`; when `offset > 0` data is written starting at byte offset `offset` in `path`.

```
>>> owproxy = protocol.proxy()
>>> owproxy.write('/10.000010EF0000/alias', b'myalias')
```

sendmess (*msgtype*, *payload*, *flags=0*, *size=0*, *offset=0*, *timeout=0*)

Send message to owserver, and blocking waits for reply.

Parameters

- **msgtype** (*int*) – message type code
- **payload** (*bytes*) – message payload
- **flags** (*int*) – message flags
- **int** (*offset*) – message size
- **int** – message offset
- **timeout** (*float*) – operation timeout (seconds)

Returns owserver return code and reply data

Return type `(int, bytes)` tuple

This is a low level method meant as direct interface to the *owserver protocol,* useful for generating messages which are not covered by the other higher level methods of this class.

This method sends a message of type `msgtype` (see *Message types*) with a given `payload` to the server; `flags` are ORed with the proxy general flags (specified in the `flags` parameter of the *`proxy()`* factory function), while `size` and `offset` are passed unchanged into the message header.

The method returns a `(retcode, data)` tuple, where `retcode` is the server return code (< 0 in case of error) and `data` the binary payload of the reply message.

```
>>> owproxy = protocol.proxy()
>>> owproxy.sendmess(protocol.MSG_DIRALL, b'/', flags=protocol.FLG_BUS_RET)
(0, b'/10.000010EF0000,/05.000005FA0100,/26.000026D90200,/01.000001FE0300,/
↪43.000043BC0400,/bus.0,/uncached,/settings,/system,/statistics,/
↪structure,/simultaneous,/alarm')
>>> owproxy.sendmess(protocol.MSG_DIRALL, b'/nonexistent')
(-1, b'')
```

class pyownet.protocol.**_PersistentProxy**

Objects of this class follow the persistent protocol, reusing the same socket connection for more than one method call. When a method is called, it firsts check for an open connection: if none is found a socket is created and bound to the owserver. All messages are sent to the server with the *FLG_PERSISTENCE* flag set; if the server grants persistence, the socket is kept open, otherwise the socket is shut down as for *_Proxy* instances. In other terms if persistence is not granted there is an automatic fallback to the non-persistent protocol.

The use of the persistent protocol is therefore transparent to the user, with an important difference: if persistence is granted by the server, a socket connection is kept open to the owserver, after the last method call. It is the responsibility of the user to explicitly close the connection at the end of a session, to avoid server timeouts.

_PersistentProxy objects have all the methods of *_Proxy* instances, plus a method for closing a connection.

close_connection()

if there is an open connection, shuts down the socket; does nothing if no open connection is present.

Note that after the call to *close_connection()* the object can still be used: in fact a new method call will open a new socket connection.

To avoid the need of explicitly calling the *close_connection()* method, *_PersistentProxy* instances support the context management protocol (i.e. the with statement.) When the `with` block is entered a socket connection is opened; the same socket connection is closed at the exit of the block. A typical usage pattern could be the following:

```
owproxy = protocol.proxy(persistent=True)

with owproxy:
    # here socket is bound to owserver
    # do work which requires to call owproxy methods
    res = owproxy.dir()
    # etc.

# here socket is closed
# do work that does not require owproxy access

with owproxy:
    # again a connection is open
    res = owproxy.dir()
    # etc.
```

In the above example, outside of the `with` blocks all socket connections to the owserver are guaranteed to be closed. Moreover the socket connection is opened when entering the block, even before the first call to a method, which could be useful for error handling.

For non-persistent connections, entering and exiting the `with` block context is a no-op.

Exceptions

Base classes

exception pyownet.protocol.**Error**
The base class for all exceptions raised by this module.

Concrete exceptions

exception pyownet.protocol.**OwnetError**
This exception is raised to signal an error return code by the owserver. This exception inherits also from the builtin OSError and follows its semantics: it sets arguments `errno`, `strerror`, and, if available, `filename`. Message errors are derived from the owserver introspection, by consulting the `/settings/return_codes/text.ALL` node.

exception pyownet.protocol.**OwnetTimeout**
This exception is raised when there is an owserver operation in progress but a given timeout period has expired. This is distinct from a low-level socket timeout which is signaled by a *ConnError*). See *Timeouts*.

exception pyownet.protocol.**ConnError**
This exception is raised when a low level socket system call fails. In fact `ConnError` simply wraps the Python OSError exception along with all its arguments, from which it inherits. In other terms it is implemented as

```
try:
    # call some socket method/function
except OSError as exc:
    raise ConnError(*exc.args)
```

Typical situations in which this exception occurs are when

- a network connection to the owserver cannot be established,
- a socket timeout occurs at the OS level.

For Python versions prior to 3.5, this exception could also be raised for an interrupted system call, see **PEP 475**[3].

exception pyownet.protocol.**ProtocolError**
This exception is raised when a successful network connection is established, but the remote server does not speak the owserver network protocol or some other error occurred during the exchange of owserver messages.

exception pyownet.protocol.**MalformedHeader**
A subclass of *ProtocolError*: raised when it is impossible to decode the reply header received from the remote owserver.

exception pyownet.protocol.**ShortRead**
A subclass of *ProtocolError*: raised when the payload received from the remote owserver is too short.

exception pyownet.protocol.**ShortWrite**
A subclass of *ProtocolError*: raised when it is impossible to send the complete payload to the remote owserver.

Exception hierarchy

The exception class hierarchy for this module is:

```
pyownet.Error
 +-- pyownet.protocol.Error
      +-- pyownet.protocol.OwnetError
      +-- pyownet.protocol.OwnetTimeout
      +-- pyownet.protocol.ConnError
```

[3] See also issue #8.

```
      +-- pyownet.protocol.ProtocolError
           +-- pyownet.protocol.MalformedHeader
           +-- pyownet.protocol.ShortRead
           +-- pyownet.protocol.ShortWrite
```

Constants

pyownet.protocol.**MAX_PAYLOAD**
: Defines the maximum number of bytes that this module is willing to read in a single message from the remote owserver. This limit is enforced to avoid security problems with malformed headers. The limit is hardcoded to 65536 bytes.[2]

Message types

These constants can by passed as the `msgtype` argument to *`_Proxy.sendmess()`* method

See also:

owserver message types

pyownet.protocol.**MSG_ERROR**

pyownet.protocol.**MSG_NOP**

pyownet.protocol.**MSG_READ**

pyownet.protocol.**MSG_WRITE**

pyownet.protocol.**MSG_DIR**

pyownet.protocol.**MSG_PRESENCE**

pyownet.protocol.**MSG_DIRALL**

pyownet.protocol.**MSG_GET**

pyownet.protocol.**MSG_DIRALLSLASH**

pyownet.protocol.**MSG_GETSLASH**

Flags

The module defines a number of constants, to be passed as the `flags` argument to *`proxy()`*. If more flags should apply, these have to be ORed together: e.g. for reading temperatures in Kelvin and pressures in Pascal, one should call:

```
owproxy = protocol.proxy(flags=FLG_TEMP_K | FLG_PRESS_PA)
```

See also:

OWFS development site: owserver flag word

general flags

pyownet.protocol.**FLG_BUS_RET**

pyownet.protocol.**FLG_PERSISTENCE**

pyownet.protocol.**FLG_ALIAS**

pyownet.protocol.**FLG_SAFEMODE**

[2] Subject to change while package is in alpha phase.

pyownet.protocol.**FLG_UNCACHED**

pyownet.protocol.**FLG_OWNET**

temperature reading flags

pyownet.protocol.**FLG_TEMP_C**

pyownet.protocol.**FLG_TEMP_F**

pyownet.protocol.**FLG_TEMP_K**

pyownet.protocol.**FLG_TEMP_R**

pressure reading flags

pyownet.protocol.**FLG_PRESS_MBAR**

pyownet.protocol.**FLG_PRESS_ATM**

pyownet.protocol.**FLG_PRESS_MMHG**

pyownet.protocol.**FLG_PRESS_INHG**

pyownet.protocol.**FLG_PRESS_PSI**

pyownet.protocol.**FLG_PRESS_PA**

sensor name formatting flags

pyownet.protocol.**FLG_FORMAT_FDI**

pyownet.protocol.**FLG_FORMAT_FI**

pyownet.protocol.**FLG_FORMAT_FDIDC**

pyownet.protocol.**FLG_FORMAT_FDIC**

pyownet.protocol.**FLG_FORMAT_FIDC**

pyownet.protocol.**FLG_FORMAT_FIC**

These flags govern the format of the 1-wire 64 bit addresses as reported by OWFS:

flag	format
FLG_FORMAT_FDIDC	10.67C6697351FF.8D
FLG_FORMAT_FDIC	10.67C6697351FF8D
FLG_FORMAT_FIDC	1067C6697351FF.8D
FLG_FORMAT_FIC	1067C6697351FF8D
FLG_FORMAT_FDI	10.67C6697351FF
FLG_FORMAT_FI	1067C6697351FF

FICD are format codes defined as below:

format	interpretation
F	family code (1 byte) as hex string
I	device serial number (6 bytes) as hex string
C	Dallas Semiconductor 1-Wire CRC (1 byte) as hex string
D	a single dot character '.'

CHAPTER 2

Indices and tables

- genindex
- search

Index

Symbols

C

D

E

F

M

O

P

R

S

W