MSL-Network Documentation

Release 1.0.0

Measurement Standards Laboratory of New Zealand

Jun 16, 2023

CONTENTS

MSL-Network uses concurrency and asynchronous programming to transfer data across a network and it is composed of three objects – a Network `Manager`, `Client`s and `Service`s.

The Network `Manager` allows for multiple `Client`s and `Service`s to connect to it and it links a `Client`'s request to the appropriate `Service` to execute the request and then the Network `Manager` sends the response from the `Service` back to the `Client`.

The Network `Manager` uses concurrency to handle requests from multiple `Client`s such that multiple requests start, run and complete in overlapping time periods and in no specific order. A `Client` can send requests synchronously or asynchronously to the Network `Manager` for a `Service` to execute. See *Concurrency and Asynchronous Programming* for more details.

JSON is used as the data format to exchange information between a `Client` and a `Service`. As such, it is possible to implement a `Client` or a `Service` in any programming language to connect to the Network `Manager`. See the *JSON Formats* section for an overview of the data format. One can also connect to the Network `Manager` from a terminal to send requests, see *Connecting from a Terminal* for more details.

CHAPTER ONE

CONTENTS

1.1 Install MSL-Network

To install MSL-Network run:

```
pip install msl-network
```

Alternatively, using the MSL Package Manager run:

```
msl install network
```

1.1.1 Compatibility

MSL-Network uses coroutines with the `async` and `await` syntax that were added in PEP492 and is compatible with Python 3.6+.

The *`Client`* and *`Service`* classes can be implemented in any programming language (and also in previous versions of Python). See the *JSON Formats* section for how the Network *`Manager`* exchanges information between a *`Client`* and a *`Service`*.

1.1.2 Dependencies

- Python 3.6+
- cryptography
- paramiko

Optional packages that can be used for (de)serializing JSON data:

- UltraJSON
- RapidJSON
- simplejson
- orjson

To use one of these external JSON packages, rather than Python's builtin `json` module, read the documentation of *`msl.network.json.Package`*.

1.2 Usage

Using MSL-Network requires a sequence of 3 steps:

1. *Start the Network Manager*
2. *Start a Service on the Network Manager*
3. *Connect to the Network Manager as a Client*

1.2.1 Start the Network Manager

The first thing to do is to start the Network `Manager`. There are 3 ways to do this.

1. From a terminal run:

   ```
   msl-network start
   ```

   Running this command will automatically perform the following default actions:

   - create a private 2048-bit, RSA key
   - create a self-signed certificate using the private key
   - create an SQLite database to store information that is used by the Network `Manager`
   - start the Network `Manager` on the default port using the TLS protocol
   - no authentication is required for `Client`'s and `Service`'s to connect to the `Manager`

   You can override the default actions, for example, use Elliptic-Curve Cryptography rather than RSA or only allow certain users to be able to connect to the `Manager`. For more details refer to the help that is available from the command line

   ```
   msl-network --help
   msl-network start --help
   ```

2. Call `run_forever()` in a script.
3. Call `run_services()` in a script. This method also starts the `Service`'s immediately after the `Manager` starts.

1.2.2 Start a Service on the Network Manager

In order to create a new Service just create a class that is a subclass of `Service` and call the `start()` method.

BasicMath Service

For example, the *BasicMath Service* is a simple (*and terribly inefficient*) `Service` that performs some basic math operations and it is included with MSL-Network.

To start the *BasicMath Service* on the `Manager` that is *running on the same computer*, run the following command in a terminal

```
python -c "from msl.examples.network import BasicMath; BasicMath().start()"
```

Note: The reason for adding the `time.sleep()` functions in the *BasicMath Service* will become evident when discussing *Asynchronous Programming*.

1.2.3 Connect to the Network Manager as a Client

Now that there is a *BasicMath Service* running on the Network `Manager` (which are both running on the same computer that the `Client` will be), we can `connect()` to the Network `Manager`

```
>>> from msl.network import connect
>>> cxn = connect(name='MyClient')
```

establish a link with the *BasicMath Service*

```
>>> bm = cxn.link('BasicMath')
```

and send a request to the *BasicMath Service*

```
>>> bm.add(1, 2)
3
```

See the Asynchronous Programming section for an example on how to send requests asynchronously.

To find out what devices are currently connected to the `Manager`, execute

```
>>> print(cxn.identities(as_string=True))
Manager[localhost:1875]
  attributes:
    identity() -> dict
    link(service: str) -> bool
  language: Python 3.9.7
  os: Windows 10 AMD64
Clients [1]:
  MyClient[localhost:63818]
    language: Python 3.9.7
    os: Windows 10 AMD64
Services [1]:
  BasicMath[localhost:63815]
    attributes:
      add(x: Union[int, float], y: Union[int, float]) -> Union[int, float]
      divide(x: Union[int, float], y: Union[int, float]) -> Union[int, float]
```

(continues on next page)

(continued from previous page)

```
      ensure_positive(x: Union[int, float]) -> bool
      euler() -> 2.718281828459045
      multiply(x: Union[int, float], y: Union[int, float]) -> Union[int,␣
↪float]
      pi() -> 3.141592653589793
      power(x: Union[int, float], n=2) -> Union[int, float]
      set_logging_level(level: Union[str, int]) -> bool
      subtract(x: Union[int, float], y: Union[int, float]) -> Union[int,␣
↪float]
    language: Python 3.9.7
    max_clients: -1
    os: Windows 10 AMD64
```

If `as_string=False`, which is the default boolean value, then the returned value would be a `dict`, rather than a `str`, containing the same information.

To disconnect from the *`Manager`*, execute

```
>>> cxn.disconnect()
```

If you only wanted to connect to the *BasicMath Service* (and no other *`Service`*s on the *`Manager`*) then you could create a *`LinkedClient`*

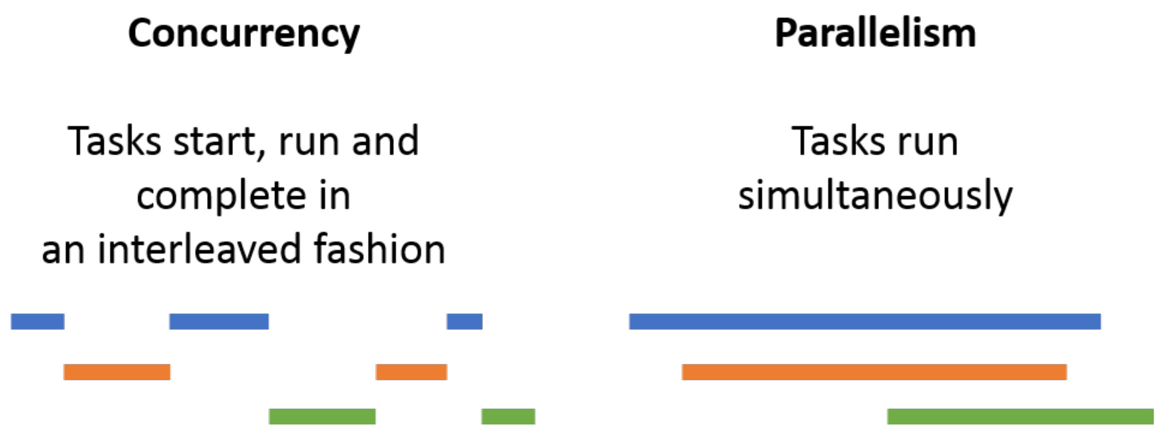
```
>>> from msl.network import LinkedClient
>>> bm = LinkedClient('BasicMath')
>>> bm.add(1, 2)
3
>>> bm.disconnect()
```

1.3 Concurrency and Asynchronous Programming

This section describes what is meant by *Concurrency* and *Asynchronous Programming*. The presentation by Robert Smallshire provides a nice overview of concurrent programming and Python's `asyncio` module.

1.3.1 Concurrency

Concurrent programming uses a single thread to execute multiple tasks in an interleaved fashion. This is different from *parallel programming* where multiple tasks can be executed at the same time.



The Network `Manager` uses *concurrent programming*. It runs in a single event loop but it can handle multiple `Client`s and `Service`s connected to it simultaneously.

When a `Client` sends a request, the `Manager` forwards the request to the appropriate `Service` and then the `Manager` waits for another event to occur. Whether the event is a reply from a `Service`, another request from a `Client` or a new device wanting to connect to the `Manager`, the `Manager` simply waits for I/O events and forwards an event to the appropriate network device when an event becomes available.

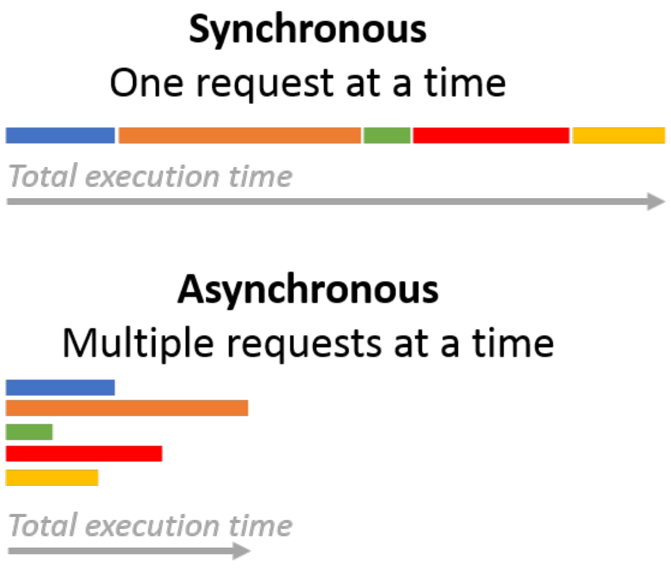
Since the `Manager` is running in a single thread it can only process one event at a single instance in time. In typical use cases, this does not inhibit the performance of the `Manager` since the `Manager` has the sole responsibility of routing requests and replies through the network and it does not actually execute a request. *There are rare situations when an administrator is making a request for the* `Manager` *to execute and in these situations the* `Manager` *would be executing the request, see* `admin_request()` *for more details.*

The `Manager` can become slow if it is (de)serializing a large JSON object or sending a large amount of bytes through the network. For example, if a reply from a `Service` is 1 GB in size and the network speed is 1 Gbps (125 MB/s) then it will take at least 8 seconds for the data to be transmitted. During these 8 seconds the `Manager` will be unresponsive to other events until it finishes sending all 1 GB of data.

If the request for, or reply from, a `Service` consumes a lot of the processing time of the `Manager` it is best to start another instance of the `Manager` on another port to host the `Service`.

1.3.2 Asynchronous Programming

A `Client` can send requests either *synchronously* or *asynchronously*. Synchronous requests are sent sequentially and the `Client` must wait to receive the reply before proceeding to send the next request. These are blocking requests where the total execution time to receive all replies is the combined sum of executing each request individually. Asynchronous requests do not wait for the reply but immediately return a `Future` instance, which is an object that is a *promise* that a result (or exception) will be available later. These are non-blocking requests where the total execution time to receive all replies is equal to the time it takes to execute the longest-running request.



Synchronous Example

The following code illustrates how to send requests *synchronously*. Before you can run this example on your own computer make sure to *Start the Network Manager* and start the *BasicMath Service*.

```
# synchronous.py
#
# This script takes about 21 seconds to run.

import time
from msl.network import connect

# Connect to the Manager (that is running on the same computer)
cxn = connect()

# Establish a link to the BasicMath Service
bm = cxn.link('BasicMath')

# Get the start time before sending the requests
t0 = time.perf_counter()

# Send all requests synchronously
# The returned object is the result of each request
add = bm.add(1, 2)
subtract = bm.subtract(1, 2)
multiply = bm.multiply(1, 2)
divide = bm.divide(1, 2)
is_positive = bm.ensure_positive(1)
power = bm.power(2, 4)

# Print the results
print(f'1+2= {add}')
print(f'1-2= {subtract}')
```

(continues on next page)

(continued from previous page)

```
print(f'1*2= {multiply}')
print(f'1/2= {divide}')
print(f'is positive? {is_positive}')
print(f'2**4= {power}')

# The total time that passed to receive all results
dt = time.perf_counter() - t0
print(f'Total execution time: {dt:.2f} seconds')

# Disconnect from the Manager
cxn.disconnect()
```

The output of the `synchronous.py` program will be:

```
1+2= 3
1-2= -1
1*2= 2
1/2= 0.5
is positive? True
2**4= 16
Total execution time: 21.06 seconds
```

The *Total execution time* value will be slightly different for you, but the important thing to notice is that executing all requests took about 21 seconds (i.e., 1+2+3+4+5+6=21 for the `time.sleep()` functions in the *BasicMath Service*) and that the returned object from each request was the value of the result.

Asynchronous Example

The following code illustrates how to send requests *asynchronously*. Before you can run this example on your own computer make sure to *Start the Network Manager* and start the *BasicMath Service*.

```
# asynchronous.py
#
# This script takes about 6 seconds to run.

import time
from msl.network import connect

# Connect to the Manager (that is running on the same computer)
cxn = connect()

# Establish a link to the BasicMath Service
bm = cxn.link('BasicMath')

# Get the start time before sending the requests
t0 = time.perf_counter()

# Create asynchronous requests by using the asynchronous=True keyword argument
# The returned object is a Future object (not the result of each request)
```

(continues on next page)

(continued from previous page)

```
add = bm.add(1, 2, asynchronous=True)
subtract = bm.subtract(1, 2, asynchronous=True)
multiply = bm.multiply(1, 2, asynchronous=True)
divide = bm.divide(1, 2, asynchronous=True)
is_positive = bm.ensure_positive(1, asynchronous=True)
power = bm.power(2, 4, asynchronous=True)

# There are different ways to gather the results of the Future objects.
# Calling result() on the Future will block until the result becomes
# available (or until the request raised an exception). Note, the
# result() method also supports a timeout argument. You can also
# register callbacks to be called when a Future is done.

# Print the results
print(f'1+2= {add.result()}')
print(f'1-2= {subtract.result()}')
print(f'1*2= {multiply.result()}')
print(f'1/2= {divide.result()}')
print(f'is positive? {is_positive.result()}')
print(f'2**4= {power.result()}')

# The total time that passed to receive all results
dt = time.perf_counter() - t0
print(f'Total execution time: {dt:.2f} seconds')

# Disconnect from the Manager
cxn.disconnect()
```

The output of the `asynchronous.py` program will be:

```
1+2= 3
1-2= -1
1*2= 2
1/2= 0.5
is positive? True
2**4= 16
Total execution time: 6.02 seconds
```

The *Total execution time* value will be slightly different for you, but the important thing to notice is that executing all requests took about 6 seconds (i.e., max(1, 2, 3, 4, 5, 6) for the `time.sleep()` functions in the *BasicMath Service*) and that the returned object from each request was a `Future` instance which we needed to get the `result()` of.

Synchronous vs Asynchronous comparison

Comparing the total execution time for the *Synchronous Example* and the *Asynchronous Example* we see that the asynchronous program is 3.5 times faster. Choosing whether to send a request synchronously or asynchronously is performed by passing in an `asynchronous=False` or `asynchronous=True` keyword argument, respectively. Also, in the synchronous example when a request is sent the object that is returned is the result of the method from the *BasicMath Service*, whereas in the asynchronous example the returned value is a `Future` object that provides the result later.

	Synchronous	Asynchronous
Total execution time	21 seconds	6 seconds
Keyword argument to invoke	asynchronous=False (default)	asynchronous=True
Returned value from request	the result	a `Future` object

1.4 JSON Formats

Information is exchanged between a *`Manager`*, a *`Client`* and a *`Service`* using JSON as the data format. The information is serialized to bytes and terminated with `"\r\n"` (a carriage return and a line feed).

A *`Client`* or a *`Service`* can be written in any programming language, but the JSON data format must adhere to the specific requirements specified below. The *`Client`* and *`Service`* must also check for the `"\r\n"` (or just the `"\n"`) byte sequence in each network packet that it receives in order to ensure that all bytes have been received or to check if multiple requests/responses are contained within the same network packet.

1.4.1 Client Format

A *`Client`* must **send a request** with the following JSON representation:

```
{
  "args": array of objects (arguments to be passed to the method of the␣
↪Manager or Service)
  "attribute": string (the name of a method or variable to access from the␣
↪Manager or Service)
  "error": false
  "kwargs": name-value pairs (keyword arguments to be passed to the method of␣
↪the Manager or Service)
  "service": string (the name of the Service, or "Manager" if the request is␣
↪for the Manager)
  "uid": string (a unique identifier of the request)
}
```

The unique identifier (uid) is only used by the *`Client`*. The *`Manager`* simply forwards the unique identifier to the *`Service`* which just includes the unique identifier in its reply. Therefore, the value can be anything that you want it to be (provided that it does not contain the `"\r\n"` sequence and it cannot be equal to `"notification"` since this is a reserved identifier). The unique identifier is useful when keeping track of which reply corresponds with which request when executing asynchronous requests.

A *Client* will also have to **send a reply** to a *Manager* during the connection procedure (i.e., when sending the *identity* of the *Client* and possibly providing a username and/or password if requested by the *Manager*).

To send a reply to the *Manager* use the following JSON representation

```
{
  "error": false (can be omitted)
  "requester": string (can be omitted)
  "result": object (the reply from the Client)
  "uid": string (can be omitted)
}
```

You only need to include the "result" name-value pair in the reply. The "error", "requester" and "uid" name-value pairs can be omitted, or anything you want, since they are not used by the *Manager* to process the reply from a *Client*. However, including these additional name-value pairs provides symmetry with the way a *Service* sends a reply to a *Manager* when there is no error.

A *Client* will **receive a reply** that is in 1 of 3 JSON representations.

Before a *Client* successfully connects to the *Manager* the *Manager* will request information about the connecting device (such as the *identity* of the device and it may check the authorization details of the connecting device).

If the bytes received represent a request from the Network *Manager* then the JSON object will be:

```
{
  "args": array of objects (arguments to be passed to the method of the␣
↪Client)
  "attribute": string (the name of a method to call from the Client)
  "error": false
  "kwargs": name-value pairs (keyword arguments to be passed to the method of␣
↪the Client)
  "requester": string (the address of the Network Manager)
  "uid": string (an empty string)
}
```

If the bytes received represent a reply from a *Service* then the JSON object will be:

```
{
  "error": false
  "requester": string (the address of the Client that made the request)
  "result": object (the reply from the Service)
  "uid": string (the unique identifier of the request)
}
```

If the bytes received represent an error then the JSON object will be:

```
{
  "error": true
  "message": string (a short description of the error)
  "requester": string (the address of the device that made the request)
  "result": null
```

(continues on next page)

(continued from previous page)

```
  "traceback": array of strings (a detailed stack trace of the error)
  "uid": string
}
```

A *Service* can also emit a notification to all *Client*'s that are *Link*ed with the *Service*. Each *Client* will **receive a notification** that has the following JSON representation

```
{
  "error": false
  "result": array (a 2-element list of [args, kwargs], e.g., [[1, 2, 3], {"x
↪": 4, "y": 5}])
  "service": string (the name of the Service that emitted the notification)
  "uid": "notification"
}
```

1.4.2 Service Format

A *Service* will **receive** data in 1 of 2 JSON representations.

If the bytes received represent an error from the Network *Manager* then the JSON object will be:

```
{
  "error": true
  "message": string (a short description of the error)
  "requester": string (the address of the Manager)
  "result": null
  "traceback": array of strings (a detailed stack trace of the error)
  "uid": string (an empty string)
}
```

If the bytes received represent a request from the *Manager* or a *Client* then the JSON object will be:

```
{
  "args": array of objects (arguments to be passed to the method of the␣
↪Service )
  "attribute": string (the name of a method or variable to access from the␣
↪Service)
  "error": false
  "kwargs": name-value pairs (keyword arguments to be passed to the method of␣
↪the Service)
  "requester": string (the address of the device that made the request)
  "uid": string (the unique identifier of the request)
}
```

A *Service* will **send a response** in 1 of 2 JSON representations.

If the *Service* raised an exception then the JSON object will be:

```
{
  "error": true
```

(continues on next page)

(continued from previous page)

```
  "message": string (a short description of the error)
  "requester": string (the address of the device that made the request)
  "result": null
  "traceback": array of strings (a detailed stack trace of the error)
  "uid": string (the unique identifier of the request)
}
```

If the `Service` successfully executed the request then the JSON object will be:

```
{
  "error": false
  "requester": string (the address of the device that made the request)
  "result": object (the reply from the Service)
  "uid": string (the unique identifier of the request)
}
```

A `Service` can also emit a notification to all `Client`'s that are `Link`ed with the `Service`. A `Service` must **emit a notification** that has the following JSON representation

```
{
  "error": false
  "result": array (a 2-element list of [args, kwargs], e.g., [[1, 2, 3], {"x
↪": 4, "y": 5}])
  "service": string (the name of the Service that emitted the notification)
  "uid": "notification"
}
```

1.5 Connecting from a Terminal

One can connect to the Network `Manager` from a terminal, e.g., using openssl s_client, to manually send requests to the Network `Manager`. So that you do not have to enter a request in the *very-specific* JSON representation of the *Client Format*, the following syntax can be used instead.

Connecting from a terminal is only convenient when connecting as a `Client`. A `Service` must enter the full JSON representation of the *Service Format* when it sends a response.

Some tips for connecting as a `Client`:

- To identify as a `Client` enter

  ```
  client
  ```

- To identify as a `Client` with the name `My Name` enter

  ```
  client My Name
  ```

- To request something from the Network `Manager` use the following format

  ```
  Manager <attribute> [<arguments>, [<keyword_arguments>]]
  ```

  For example, to request the `identity` of the Network `Manager` enter

```
Manager identity
```

or, as a shortcut for requesting the *identity* of the *Manager*, you only need to enter

```
identity
```

To check if a user with the name `n.bohr` exists in the database of registered users enter

```
Manager users_table.is_user_registered n.bohr
```

Note: Most requests that are for the Network *Manager* to execute require that the request comes from a *Client* that is connected to the Network *Manager* as an administrator. Your login credentials will be checked (requested from you) before the Network *Manager* executes the request. See the `user` command in *MSL-Network CLI Documentation* for more details on how to become an administrator.

- To request something from a *Service* use the following format

```
<service> <attribute> [<arguments>, [<keyword_arguments>]]
```

Attention: Although you can send requests to a *Service* in the following manner there is no way to block the request if the *Service* has already met the restriction for the maximum number of *Client*'s that can be linked with the *Service* to send requests to it. Therefore, you should only do the following if you are certain that the *Service* has not reached its maximum *Client* limit. To test if this *Client* limit has been reached enter `link <service>`, for example, `link BasicMath` and see if you get a `PermissionError` in the response before you proceed to send requests to the *Service*.

For example, to request the addition of two numbers from the *BasicMath Service* enter

```
BasicMath add 4 10
```

or

```
BasicMath add x=4 y=10
```

To request the concatenation of two strings from a `ModifyString.concat(s1, s2)` *Service*, but with the `ModifyString` *Service* being named `String Editor` on the Network *Manager* enter

```
"String Editor" concat s1="first string" s2="second string"
```

- To disconnect from the Network *Manager* enter

```
disconnect
```

or

```
exit
```

1.6 Python Examples

The following examples illustrate some ideas on how one could use MSL-Network.

1. *Digital Multimeter*
2. *Additional (Runnable) Examples*
3. RPi-SmartGadget – Uses a Raspberry Pi to communicate with a Sensirion SHTxx sensor.

1.6.1 Digital Multimeter

This example shows how a digital multimeter that has a non-Ethernet interface, e.g., GPIB or RS232, can be controlled from any computer that is on the network. It uses the MSL-Equipment package to connect to the digital multimeter and MSL-Network to enable the digital multimeter as a `Service` on the network. This example is included with MSL-Network when it is installed, but since it requires additional hardware (a digital multimeter) it can only be run if the hardware is attached to the computer.

The first task to do is to *Start the Network Manager* on the same computer that the digital multimeter is physically connected to (via a GPIB cable or a DB9 cable). Next, on the same computer, copy and paste the following script to a file, edit the equipment record used by MSL-Equipment for the information relevant to your DMM (e.g., the COM#, GPIB address) and then run the script to start the digital multimeter `Service`.

```
"""
Example showing how a digital multimeter that has a non-Ethernet interface
(e.g., GPIB or RS232) can be controlled from any computer that is on the␣
↪network.
"""
from msl.equipment import ConnectionRecord
from msl.equipment import EquipmentRecord

from msl.network import Service


class DigitalMultimeter(Service):

    def __init__(self):
        """Initialize the communication with the digital multimeter.

        This script must be run on a computer that the multimeter is
        physically connected to.
        """

        # Initialize the Service. Set the name of the DigitalMultimeter␣
↪Service,
        # as it will appear on the Network Manager, to be 'Hewlett Packard␣
↪34401A'
```

(continues on next page)

(continued from previous page)

```
        # and specify that only 1 Client on the network can control the␣
↪digital
        # multimeter at any instance in time. Once the Client disconnects from
        # the Network Manager another Client would then be able to link with␣
↪the
        # DigitalMultimeter Service to control the digital multimeter.
        super().__init__(name='Hewlett Packard 34401A', max_clients=1)

        # Connect to the digital multimeter
        # (see MSL-Equipment for more details)
        record = EquipmentRecord(
            manufacturer='HP',
            model='34401A',
            connection=ConnectionRecord(
                address='COM4',  # RS232 interface
                backend='MSL',
            )
        )
        self._dmm = record.connect()

    def write(self, command: str) -> None:
        """Write a command to the digital multimeter.

        Parameters
        ----------
        command : str
            The command to write.
        """
        self._dmm.write(command)

    def read(self) -> str:
        """Read the response from the digital multimeter.

        Returns
        -------
        str
            The response.
        """
        return self._dmm.read().rstrip()

    def query(self, command: str) -> str:
        """Query the digital multimeter.

        Performs a write then a read.

        Parameters
        ----------
        command : str
            The command to write.
```

(continues on next page)

(continued from previous page)

```
        Returns
        -------
        str
            The response.
        """
        return self._dmm.query(command).rstrip()


if __name__ == '__main__':
    # Initialize and start the DigitalMultimeter Service
    dmm_service = DigitalMultimeter()
    dmm_service.start()
```

With the `DigitalMultimeter` *`Service`* running you can execute the following commands on another computer that is on the same network as the *`Manager`* in order to interact with the digital multimeter from the remote computer.

Connect to the *`Manager`* by specifying the hostname (or IP address) of the computer that the *`Manager`* is running on

```
>>> from msl.network import connect
>>> cxn = connect(host='the hostname or IP address of the computer that the␣
↪Manager is running on')
```

Since the name of the `DigitalMultimeter` *`Service`* was specified to be `'Hewlett Packard 34401A'`, we must link with the correct name of the *`Service`*

```
>>> dmm = cxn.link('Hewlett Packard 34401A')
```

Tip: The process of establishing a connection to a *`Manager`* and linking with a *`Service`* can also be done in a single line. A *`LinkedClient`* exists for this purpose. This can be useful if you only want to link with a single *`Service`*.

```
>>> from msl.network import LinkedClient
>>> dmm = LinkedClient('Hewlett Packard 34401A', host='hostname or IP address␣
↪of the Manager')
```

Now we can send `write`, `read` or `query` commands to the digital multimeter

```
>>> dmm.query('MEASURE:VOLTAGE:DC?')
'-6.23954727E-02'
```

When you are finished sending requests to the *`Manager`* you should disconnect from the *`Manager`*. This will allow other *`Client`*'s to be able to control the digital multimeter.

```
>>> cxn.disconnect()
```

1.6.2 Additional (Runnable) Examples

The following `Service`'s are included with MSL-Network. To start any of these `Service`'s, first make sure that you *Start the Network Manager*, and then run the following command in a terminal. For this example, the *Echo Service* is running

```
python -c "from msl.examples.network import Echo; Echo().start()"
```

You can then send requests to the *Echo Service*

```
>>> from msl.network import connect
>>> cxn = connect()
>>> e = cxn.link('Echo')
>>> e.echo('hello')
[['hello'], {}]
>>> e.echo('world!', x=7, array=list(range(10)))
[['world!'], {'x': 7, 'array': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]}]
>>> cxn.disconnect()
```

Echo Service

```
"""
Example echo Service.

Before running this module ensure that the Network Manager is running on the
same computer, i.e., run the following command in a terminal

msl-network start

then run this module to connect to the Manager as a Service.

After the Echo Service starts you can connect to the Manager as a Client,
link with the Echo Service and then send requests, e.g.,

from msl.network import connect
cxn = connect()
e = cxn.link('Echo')
args, kwargs = e.echo(1, 2, x='hello', y='world')
"""
from msl.network import Service


class Echo(Service):

    @staticmethod
    def echo(*args, **kwargs):
        return args, kwargs


if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    service = Echo()
    service.start()
```

BasicMath Service

```
"""
Example Service for illustrating the difference between synchronous and
asynchronous requests.

Before running this module ensure that the Network Manager is running on the
same computer, i.e., run the following command in a terminal

msl-network start

then run this module to connect to the Manager as a Service.

After the BasicMath Service starts you can connect to the Manager as a Client,
link with the BasicMath Service and then send requests, e.g.,

from msl.network import connect
cxn = connect()
bm = cxn.link('BasicMath')
value = bm.add(1, 2)
"""
import time
from typing import Union

from msl.network import Service

number = Union[int, float]


class BasicMath(Service):

    euler = 2.718281828459045

    @property
    def pi(self) -> float:
        return 3.141592653589793

    def add(self, x: number, y: number) -> number:
        time.sleep(1)
        return x + y

    def subtract(self, x: number, y: number) -> number:
        time.sleep(2)
        return x - y
```

(continues on next page)

(continued from previous page)

```
    def multiply(self, x: number, y: number) -> number:
        time.sleep(3)
        return x * y

    def divide(self, x: number, y: number) -> number:
        time.sleep(4)
        return x / float(y)

    def ensure_positive(self, x: number) -> bool:
        time.sleep(5)
        if x < 0:
            raise ValueError('The value is < 0')
        return True

    def power(self, x: number, n=2) -> number:
        time.sleep(6)
        return x ** n


if __name__ == '__main__':
    import logging

    # Optional: allows for "info" log messages to be visible on the Service
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s [%(levelname)-5s] %(message)s',
    )

    service = BasicMath()
    service.start()
```

MyArray Service

```
"""
Example Service for generating and manipulating arrays. This example
illustrates how to interface a LabVIEW program with MSL-Network.

Before running this module ensure that the Network Manager is running on the
same computer, i.e., run the following command in a terminal

msl-network start

then run this module to connect to the Manager as a Service.

After the MyArray Service starts you can connect to the Manager as a Client,
link with the MyArray Service and then send requests, e.g.,

from msl.network import connect
```

(continues on next page)

(continued from previous page)

```
cxn = connect()
my_array = cxn.link('MyArray')
linspace = my_array.linspace(0, 1)
"""
from typing import List, Union

from msl.network import Service

number = Union[int, float]
Vector = List[float]


class MyArray(Service):

    @staticmethod
    def linspace(start: number, stop: number, n=100) -> List[float]:
        """Return evenly-spaced numbers over a specified interval."""
        dx = (stop-start)/float(n-1)
        return [start+i*dx for i in range(int(n))]

    @staticmethod
    def scalar_multiply(scalar: number, data: Vector) -> Vector:
        """Multiply every element in `data` by a number."""
        return [element*scalar for element in data]


if __name__ == '__main__':
    service = MyArray()
    service.start()
```

Heartbeat Service

```
"""
Example Service that emits notifications to all linked Clients. This example
also shows how to add a task to the event loop of the Service.

Before running this module ensure that the Network Manager is running on the
same computer, i.e., run the following command in a terminal

msl-network start

then run this module to connect to the Manager as a Service.

After the Heartbeat Service starts you can connect to the Manager as a Client,
link with the Heartbeat Service, handle notifications from the Service and␣
↪also
send requests, e.g.,
```

(continues on next page)

(continued from previous page)

```
import types
from msl.network import connect

def print_notification(self, *args, **kwargs):
    print(f'The {self.service_name} Service emitted', args, kwargs)

cxn = connect()
heartbeat = cxn.link('Heartbeat')
heartbeat.notification_handler = types.MethodType(print_notification,␣
↪heartbeat)

# some time later

heartbeat.reset()
"""
import asyncio

from msl.network import Service


class Heartbeat(Service):

    def __init__(self):
        """A Service that emits a counter value."""
        super(Heartbeat, self).__init__()
        self._sleep = 1.0
        self._counter = 0
        self._alive = True

    def kill(self) -> None:
        """Stop emitting the heartbeat."""
        self._alive = False

    def reset(self) -> None:
        """Reset the heartbeat counter."""
        self._counter = 0

    def set_heart_rate(self, beats_per_second: int) -> None:
        """Change the rate that the value of the counter is emitted."""
        self._sleep = 1.0 / float(beats_per_second)

    def shutdown_handler(self) -> None:
        """Called when the connection to the Manager is closed."""
        self._alive = False

    async def emit(self) -> None:
        """This coroutine is also run in the event loop."""
        while self._alive:
            self.emit_notification(self._counter)
```

(continues on next page)

(continued from previous page)

```
            self._counter += 1
            await asyncio.sleep(self._sleep)


if __name__ == '__main__':
    # Initialize the Service
    service = Heartbeat()

    # Add a task to the event loop of the Service
    service.add_tasks(service.emit())

    # Start the Service
    service.start()
```

1.7 Non-Python Examples

Since information is sent using the *JSON format* across the network the `Client` and `Service` classes can be implemented in any programming language.

1.7.1 LabVIEW

The following illustrates how to use LabVIEW to send requests as a Client and receive requests as a Service. The source code is available to download from the repository. The VI's have been saved with LabVIEW 2010. The LabVIEW code uses the i3 JSON Toolkit, which is bundled with the code in the repository, to (de)serialize JSON data.

Attention: The *asynchronous* aspect of MSL-Network is not implemented in the VI's.

The first step is to *Start the Network Manager*. Since LabVIEW does not natively support the TLS protocol you must start the Network `Manager` with the `--disable-tls` flag, and, to simplify the examples below, do not use any authentication, i.e., run

```
msl-network start --disable-tls
```

The *hostname* and *port* number that the Network `Manager` is running on will be displayed. These values will need to be entered in the front panel of the VI's shown below.

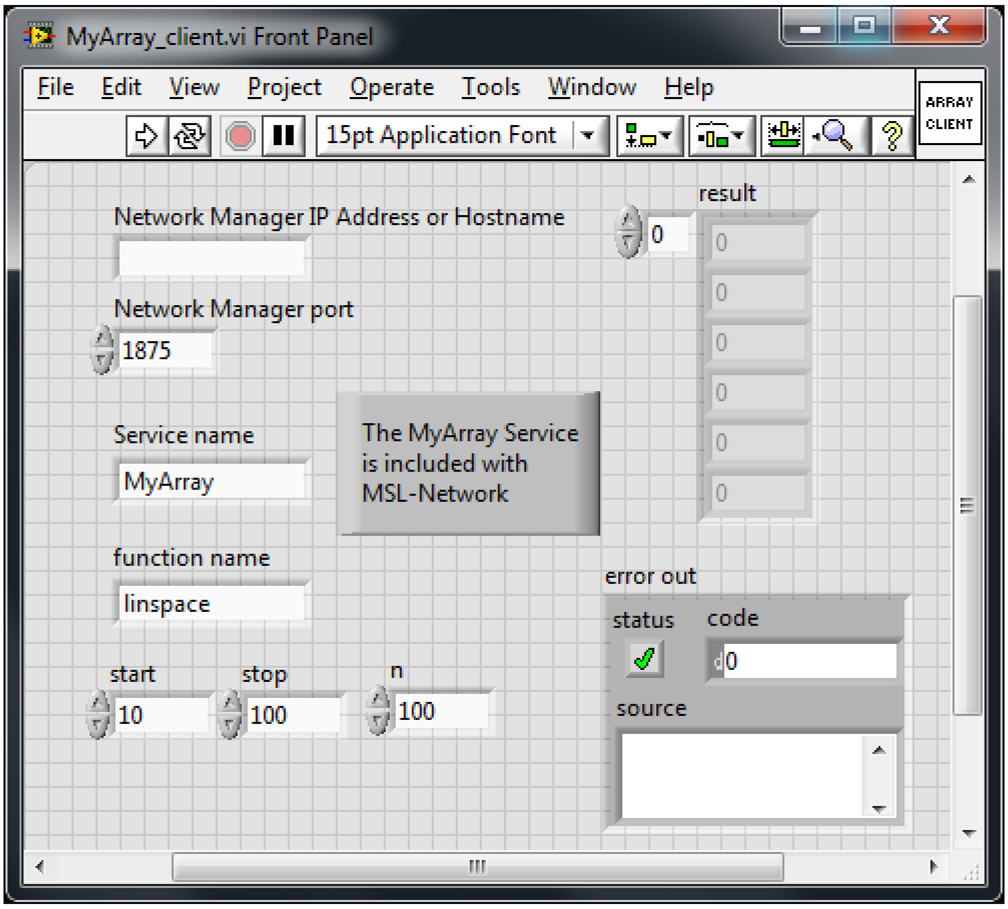
```
... [INFO ] msl.network - Network Manager running on <hostname>:<port> (TLS␣
↪DISABLED)
```

Client

The following shows how to send a request to the *MyArray Service*. Before running `MyArray_client.vi` make sure that the *MyArray Service* is running on the Network `Manager`

```
python -c "from msl.examples.network import MyArray; MyArray().start(disable_
↪tls=True)"
```

On the front panel of `MyArray_client.vi` you need to enter the *hostname* and *port* values that the Network `Manager` is running on (see above). The *Service name* and *function name* values on the front panel do not need to be changed for this example. By changing the values of the *start*, *stop* and *n* parameters the *result* array will be populated when you run the VI.



Network Manager IP Address or Hostname

Network Manager port: 1875

Service name: MyArray

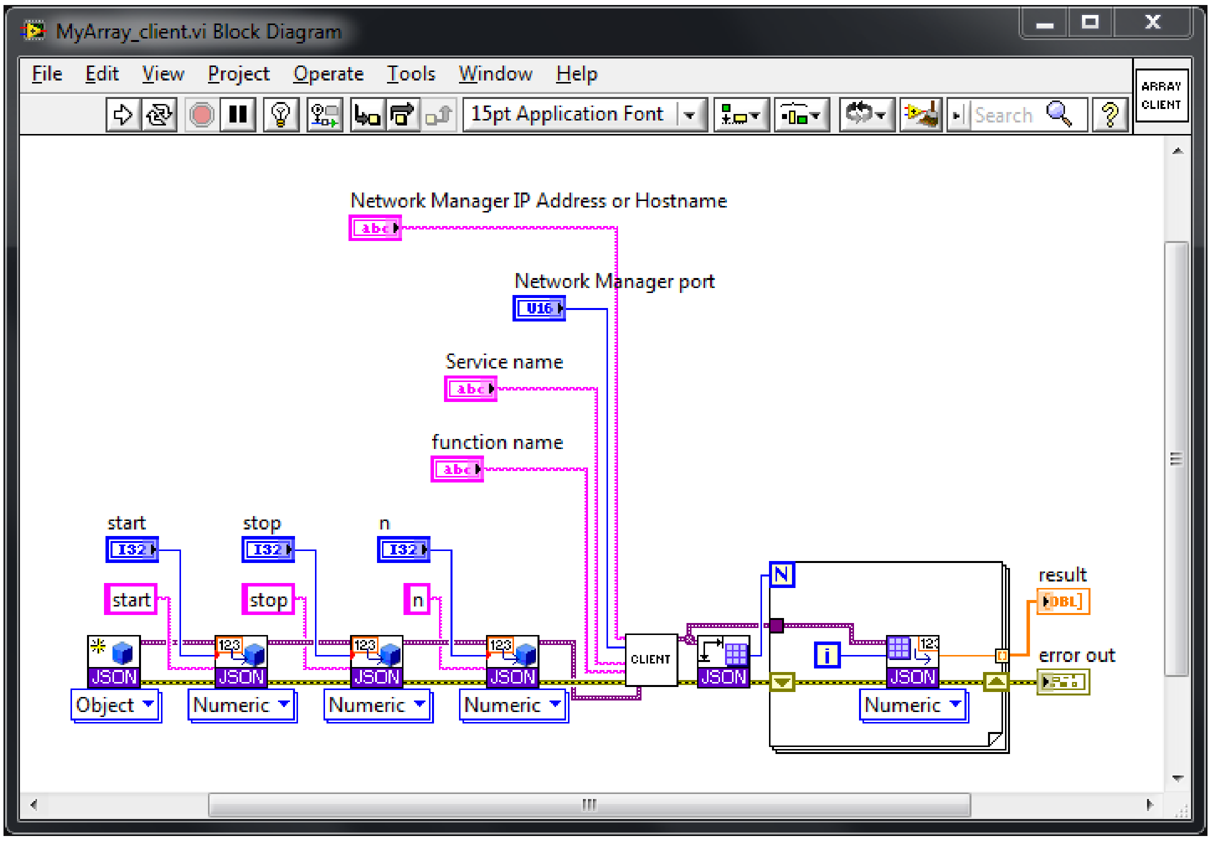
function name: linspace

start: 10 stop: 100 n: 100

The MyArray Service is included with MSL-Network

result: 0 | 0 | 0 | 0 | 0 | 0 | 0

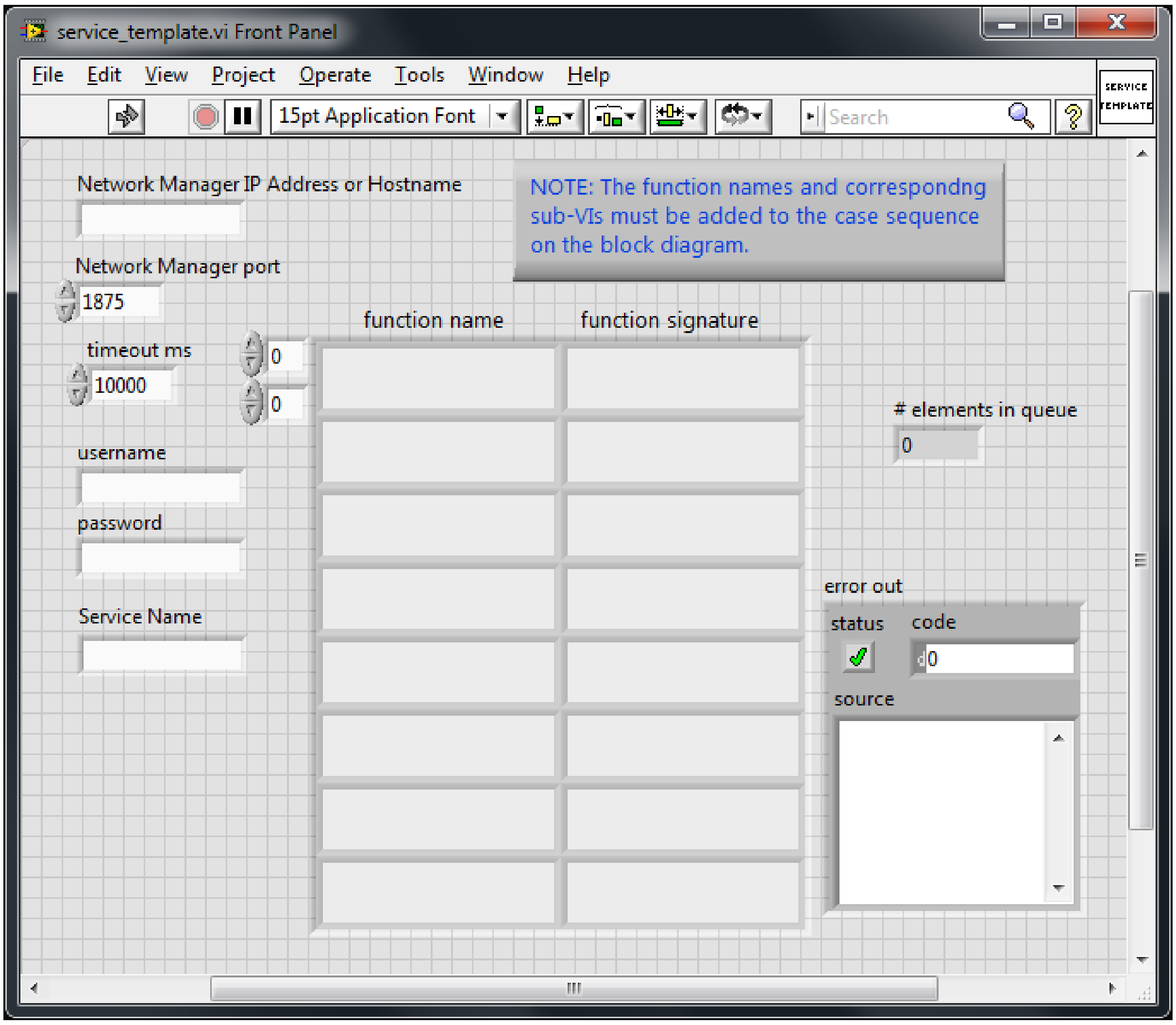
error out
status: ✓ code: 0
source



Service

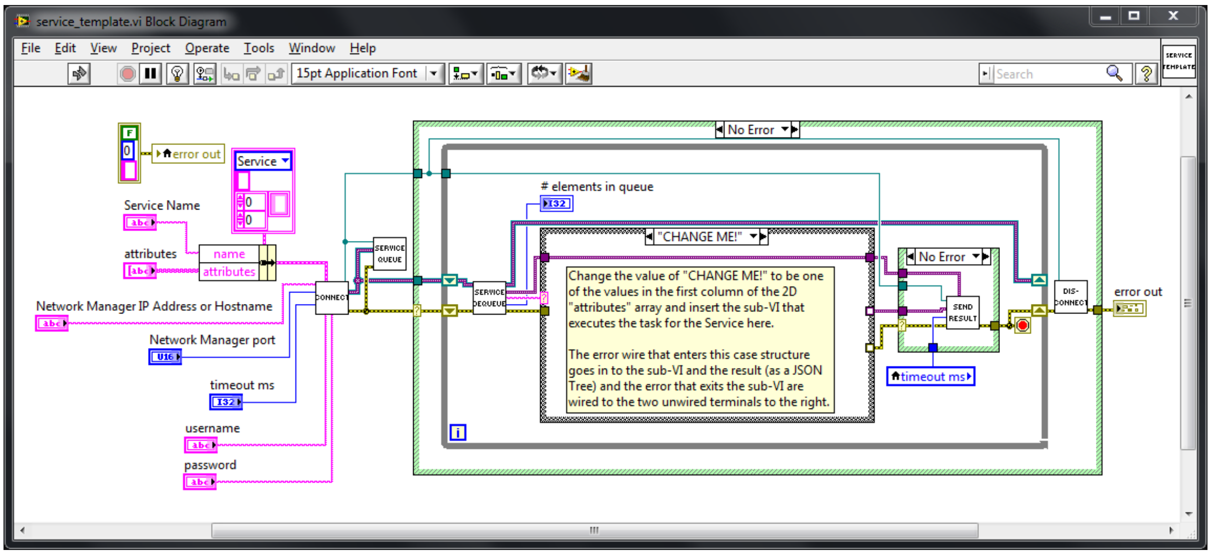
The `service_template.vi` file is a template to use for creating a new Service. The controls on the front panel of the VI are as follows:

- *Network Manager IP Address or Hostname* and *Network Manager port*: The *hostname* and *port* values that the Network `Manager` is running on (see above).
- *timeout ms*: The maximum number of milliseconds to wait to connect to the Network `Manager`.
- *username* and *password*: Since the Network `Manager` can be started using different types of authentication for a Client or Service to be allowed to connect to it you can specify the values here. If the *username* and/or *password* values are not specified and the Network `Manager` requires these values for the connection then LabVIEW will prompt you for these values.
- *Service Name*: The name of your Service as it will appear on the Network `Manager`.
- *function name* and *function signature*: These are used to let a Client know what functions your Service provides, what input parameters are needed for each function and what each function returns. For more details see the comments in the `Service -> attributes` section in the `identity()` method.



NOTE: The function names and correspondng sub-VIs must be added to the case sequence on the block diagram.

The case sequence on the block diagram needs to be updated for each function that your Service provides

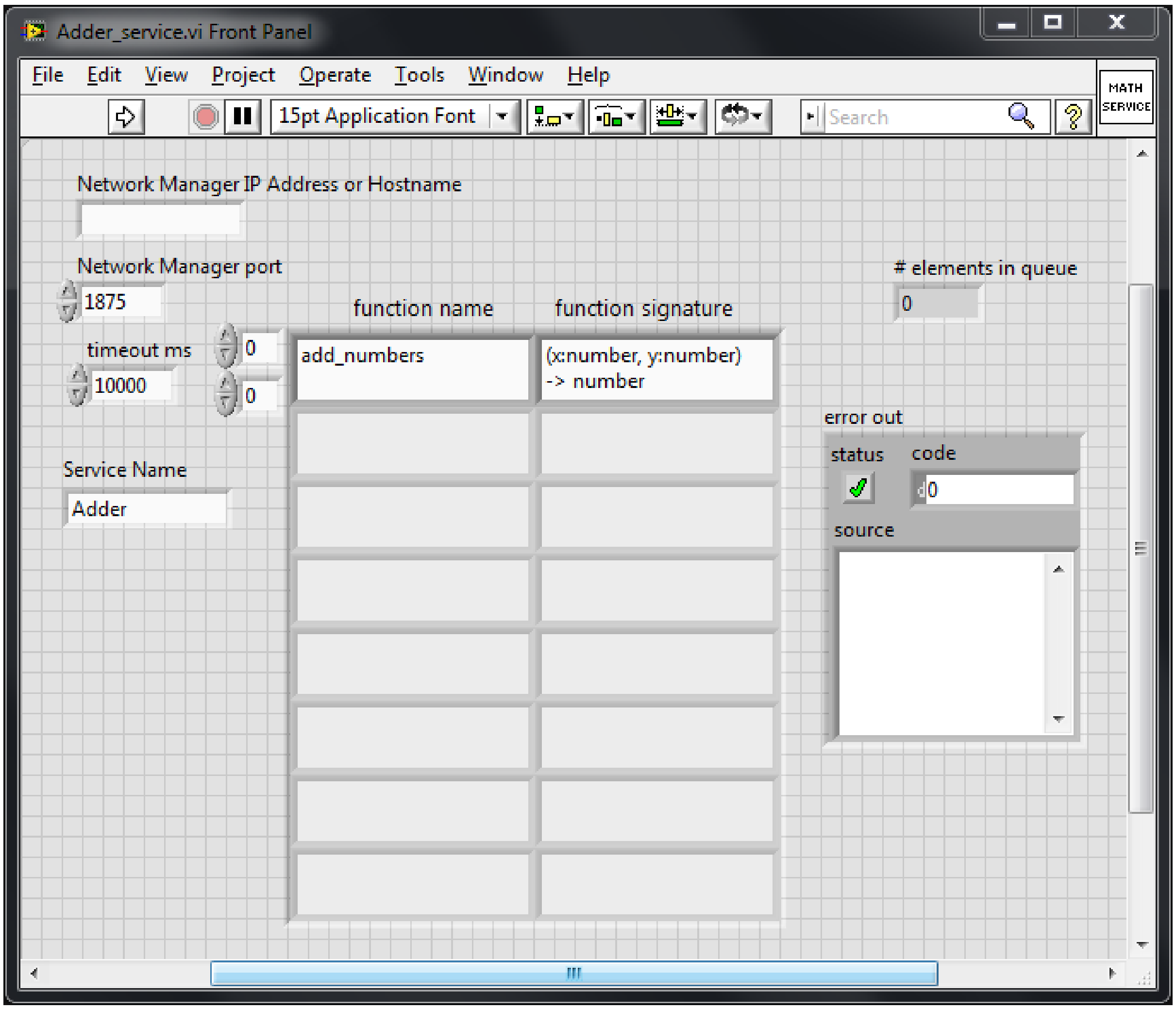


Change the value of "CHANGE ME!" to be one of the values in the first column of the 2D "attributes" array and insert the sub-VI that executes the task for the Service here.

The error wire that enters this case structure goes in to the sub-VI and the result (as a JSON Tree) and the error that exits the sub-VI are wired to the two unwired terminals to the right.

Adder Service

As a particular example of implementing a Service in LabVIEW the following VI shows an *Adder* Service. This Service has a function called *add_numbers* that takes two numbers as inputs, *x* and *y*, and returns the sum.


Adder_service.vi Front Panel
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
MATH SERVICE

Network Manager IP Address or Hostname

Network Manager port: 1875

timeout ms: 10000

0

0

Service Name: Adder

function name	function signature
add_numbers	(x:number, y:number) -> number

elements in queue: 0

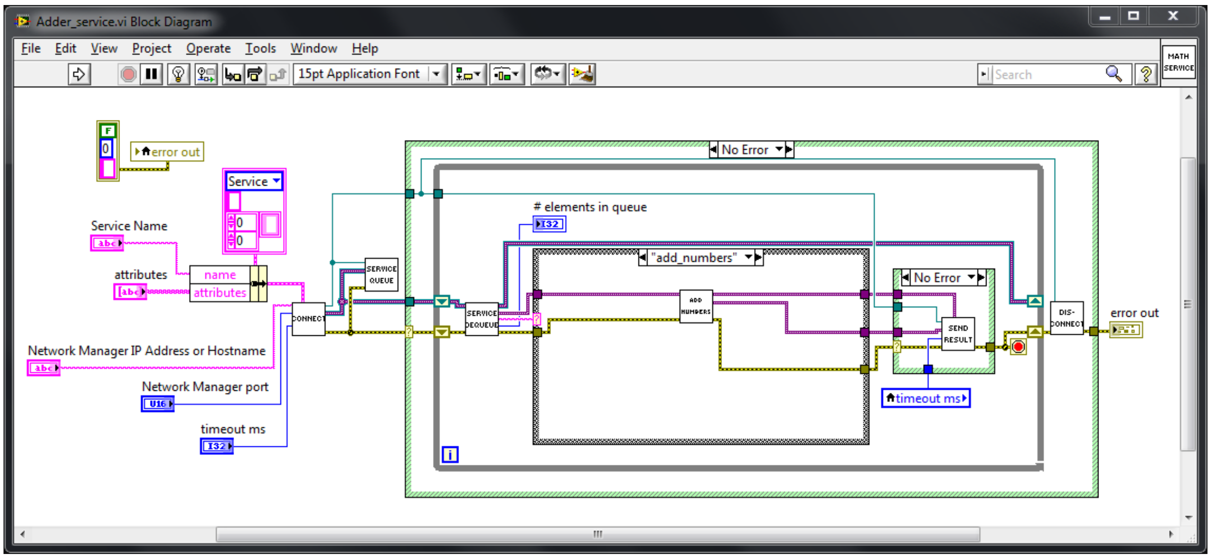
error out

status: ✓

code: 0

source

Note that the name of the *add_numbers* function is specified on the front panel (which lets Clients know that this function exists) and in the case structure on the block diagram (which processes a Client's request).



Run `Adder_service.vi` to start the *Adder* Service and then on another computer you can send a request to the *Adder* Service

```
>>> from msl.network import connect
>>> cxn = connect(host='the hostname or IP address of the Manager', disable_
↪tls=True)
```

establish a link with the *Adder* Service

```
>>> adder = cxn.link('Adder')
```

and send a request to the *Adder* Service

```
>>> adder.add_numbers(x=1.2, y=3.4)
4.6
```

Disconnect from the Network `Manager` when you are finished

```
>>> cxn.disconnect()
```

1.8 Starting a Service from another computer

Suppose that you wanted to start a `Service` on a remote computer, for example, a Raspberry Pi, from another computer that is on the same network as the Pi.

Your package has the following structure:

```
mypackage/
    mypackage/
        __init__.py
        my_client.py
        rpi_service.py
    setup.py
```

with `setup.py` as

```
from setuptools import setup

setup(
    name='mypackage',
    version='0.1.0',
    packages=['mypackage'],
    install_requires=['msl-network'],
    entry_points={
        'console_scripts': [
            'mypackage = mypackage:start_service_on_rpi',
        ],
    },
)
```

`__init__.py` as

```
from msl.network import run_services, ssh, LinkedClient

from .rpi_service import RPiService
from .my_client import MyClient

def connect(*, host='raspberrypi', rpi_password=None, timeout=10, **kwargs):
    # you will need to update the `console_script_path` value below
    # when you implement the code in your own program since this is a unique␣
↪path
    # that is defined as the path where the mypackage executable is located␣
↪on the Pi
    console_script_path = '/home/pi/.local/bin/mypackage'
    ssh.start_manager(host, console_script_path, ssh_username='pi',
                      ssh_password=rpi_password, timeout=timeout, **kwargs)

    # create a Client that is linked with a Service of your choice
    # in this case it is the RPiService
    kwargs['host'] = host
    return MyClient('RPiService', **kwargs)

def start_service_on_rpi():
    # this function gets called from your `console_scripts` definition in␣
↪setup.py
    kwargs = ssh.parse_console_script_kwargs()
    if kwargs.get('auth_login', False) and ('username' not in kwargs or
↪'password' not in kwargs):
        raise ValueError(
            'The Manager is using a login for authentication but RPiService '
            'does not know the username and password to use to connect to the␣
↪Manager'
        )
    run_services(RPiService(), **kwargs)
```

`rpi_service.py` as

```python
from msl.network import Service

class RPiService(Service):

    def __init__(self):
        super(RPiService, self).__init__()

    def shutdown_service(self, *args, **kwargs):
        # Implementing this method allows for the RPiService to be
        # shut down remotely by MyClient. MyClient can also include
        # *args and **kwargs to the shutdown_service() method.
        # If there is nothing that needs to be performed before the
        # RPiService shuts down then just return None.
        # After the shutdown_service() method returns, the RPiService
        # will automatically wait for all Futures that it is currently
        # executing to either finish or to be cancelled before the
        # RPiService disconnects from the Network Manager.
        pass

    def add_numbers(self, a, b, c, d):
        return a + b + c + d

    def power(self, a, n=2):
        return a ** n
```

and `my_client.py` as

```python
from msl.network import LinkedClient

class MyClient(LinkedClient):

    def __init__(self, service_name, **kwargs):
        super(MyClient, self).__init__(service_name, **kwargs)

    def disconnect(self):
        # We override the `disconnect` method because we want to shut
        # down the RPiService and the Network Manager when MyClient
        # disconnects from the Raspberry Pi. Not every Service will
        # allow a Client to shut it down. However, we have decided to
        # design mypackage in a particular way that MyClient is
        # intended to be the only Client connected to the Manager and
        # when MyClient is done communicating with the RPiService then
        # both the Manager and the Service shut down. The Client can
        # also include *args and **kwargs in the shutdown_service()
        # request, but we don't use them in this example.
        self.shutdown_service()
        super(MyClient, self).disconnect()

    def service_error_handler(self):
        # We can override this method to handle the situation if
```

(continues on next page)

(continued from previous page)

```
        # there is an error on the Service. In general, if a Service
        # raises an exception you wouldn't want it to shut
        # down because you would have to manually restart it. Especially
        # if other Clients are requesting information from that Service.
        # However, for mypackage we want everything to shut down
        # (RPiService, MyClient and the Manager) when any one of them
        # raises an exception.
        self.disconnect()
```

To create a source distribution of `mypackage` run the following in the root folder of your package directory

```
python setup.py sdist
```

This will create a file `dist/mypackage-0.1.0.tar.gz`. Copy this file to the Raspberry Pi.

The following libraries are needed to install the cryptography package from source on the Raspberry Pi.

```
sudo apt install build-essential libssl-dev libffi-dev python3-dev
```

Note: It is recommended to install `mypackage` in a virtual environment if you are familiar with them. However, in what follows we show how to install `mypackage` without using a virtual environment for simplicity.

Install `mypackage-0.1.0.tar.gz` on the Raspberry Pi using

```
pip3 install mypackage-0.1.0.tar.gz
```

In addition, install `mypackage-0.1.0.tar.gz` on another computer.

Finally, on the *'another'* computer you would perform the following. This would start the Network *Manager* on the Raspberry Pi, start the `RPiService`, connect to the *Manager* and *link()* with `RPiService`.

You may have to change the value of *host* for your Raspberry Pi. The following example assumes that the hostname of the Raspberry Pi is `raspberrypi`.

```
>>> from mypackage import connect
>>> rpi = connect(host='raspberrypi')
>>> rpi.add_numbers(1, 2, 3, 4)
10
>>> rpi.power(4)
16
>>> rpi.power(5, n=3)
125
```

When you are done sending requests to `RPiService` you call the `disconnect` method which will shut down the `RPiService` and the Network *Manager* that are running on the Raspberry Pi and disconnect `MyClient` from the Pi.

```
>>> rpi.disconnect()
```

Tip: Suppose that you get the following error

```
>>> rpi = connect(host='raspberrypi')
...
[Errno 98] error while attempting to bind on address ('::', 1875, 0, 0):␣
↪address already in use
```

This means that there is probably a *Manager* already running on the Raspberry Pi at port 1875. You have four options to solve this problem using MSL-Network.

(1) Start another *Manager* on a different port

```
>>> rpi = connect(host='raspberrypi', port=1876)
```

(2) Connect to the *Manager* and shut it down gracefully; however, this requires that you are an administrator of that *Manager*. See the `user` command in *MSL-Network CLI Documentation* for more details on how to create a user that is an administrator.

```
>>> from msl.network import connect, constants
>>> cxn = connect(host='raspberrypi')
>>> cxn.admin_request(constants.SHUTDOWN_MANAGER)
```

(3) Kill the *Manager*

```
>>> from msl.network import ssh
>>> ssh_client = ssh.connect('pi@raspberrypi')
>>> out = ssh.exec_command(ssh_client, 'ps aux | grep mypackage')
>>> print('\n'.join(out))
pi   1367  0.1  2.2  63164 21380 pts/0  Sl+  12:21  0:01 /usr/bin/python3 .
↪local/bin/mypackage
pi   4341  0.0  0.2   4588  2512 ?      Ss   12:30  0:00 bash -c ps aux | grep␣
↪mypackage
pi   4343  0.0  0.0   4368   540 ?      S    12:30  0:00 grep mypackage
>>> ssh.exec_command(ssh_client, 'sudo kill -9 1367')
[]
>>> ssh_client.close()
```

(4) Reboot the remote computer

```
>>> from msl.network import ssh
>>> ssh_client = ssh.connect('pi@raspberrypi')
>>> ssh.exec_command(ssh_client, 'sudo reboot')
[]
>>> ssh_client.close()
```

1.9 MSL-Network CLI Documentation

The follow commands summarize the various ways to use MSL-Network from a terminal.

1.9.1 msl.network.cli_certdump module

Command line interface for the `certdump` command.

To see the help documentation, run the following command in a terminal:

```
msl-network certdump --help
```

Dumps the details of a PEM certificate.

The `certdump` command is similar to the openssl command to get the details of a certificate:

```
openssl x509 -in certificate.crt -text -noout
```

Examples:

```
# dump the details to the terminal
msl-network certdump /path/to/cert.pem

# dump the details to a file
msl-network certdump /path/to/cert.pem --out dump.txt
```

See Also:

```
msl-network certgen
```

`msl.network.cli_certdump.`**`add_parser_certdump`**`(`*parser*`)`

Add the `certdump` command to the *parser*.

`msl.network.cli_certdump.`**`execute`**`(`*args*`)`

Executes the `certdump` command.

1.9.2 msl.network.cli_certgen module

Command line interface for the `certgen` command.

To see the help documentation, run the following command in a terminal:

```
msl-network certgen --help
```

Generate a self-signed PEM certificate.

The certificate uses the hostname of the computer that this command was executed on as the Common Name and as the Issuer Name.

The `certgen` command is similar to the openssl command to generate a self-signed certificate from a pre-existing private key:

```
openssl req -key private.key -new -x509 -days 365 --out certificate.crt
```

Examples:

```
# create a default certificate using the default private key
# and save it to the default directory
msl-network certgen

# create a certificate using the specified key and
# save the certificate to the specified file
msl-network certgen --key-file /path/to/key.pem /path/to/cert.pem
```

See Also:

```
msl-network keygen
msl-network certdump
```

`msl.network.cli_certgen.`**`add_parser_certgen`**`(`*parser*`)`

Add the `certgen` command to the *parser*.

`msl.network.cli_certgen.`**`execute`**`(`*args*`)`

Executes the `certgen` command.

1.9.3 msl.network.cli_delete module

Command line interface for the `delete` command.

New in version 1.0.

To see the help documentation, run the following command in a terminal:

```
msl-network delete --help
```

Delete files that are created by MSL-Network.

Can remove the database, log files, certificates and/or keys.

Examples:

```
# delete all files that are created by MSL-Network
msl-network delete --all

# delete all log files
msl-network delete --logs
```

`msl.network.cli_delete.`**`add_parser_delete`**`(`*parser*`)`

Add the `delete` command to the *parser*.

`msl.network.cli_delete.`**`execute`**`(`*args*`)`

Executes the `delete` command.

1.9.4 msl.network.cli_hostname module

Command line interface for the `hostname` command.

To see the help documentation, run the following command in a terminal:

```
msl-network hostname --help
```

Add/remove hostname(s) into/from the table in the database.

The Network Manager can be started with the option to use trusted devices (based on the hostname of the connecting device) as the authorisation check for a Client or Service to be able to connect to the Network Manager.

Each hostname in the table is considered as a trusted device and therefore the device can connect to the Network Manager.

To use trusted hostnames as the authentication check, start the Network Manager with the `--auth-hostname` flag:

```
msl-network start --auth-hostname
```

Examples:

```
# add 'TheHostname' as a trusted device in the default database
msl-network hostname add TheHostname

# add 'TheHostname' and 'OtherHostname' as trusted devices
msl-network hostname add TheHostname OtherHostname

# remove 'OtherHostname' from the database of trusted devices
msl-network hostname remove OtherHostname

# add 'TheHostname' to a specific database
msl-network hostname add TheHostname --database /path/to/database.db

# list all trusted hostnames
msl-network hostname list
```

`msl.network.cli_hostname.`**`add_parser_hostname`**(*parser*)

Add the `hostname` command to the *parser*.

`msl.network.cli_hostname.`**`execute`**(*args*)

Executes the `hostname` command.

1.9.5 msl.network.cli_keygen module

Command line interface for the `keygen` command.

To see the help documentation, run the following command in a terminal:

```
msl-network keygen --help
```

Generate a private key to digitally sign a PEM certificate.

The `keygen` command is similar to the openssl command:

```
openssl req -newkey rsa:2048 -nodes -keyout key.pem
```

Examples:

```
# create a default private key (RSA, 2048-bit, unencrypted)
# and save it to the default directory
msl-network keygen

# create a 3072-bit, encrypted private key using the DSA algorithm
msl-network keygen dsa --size 3072 --password WhatEVER you wAnt!
```

See Also:

```
msl-network certgen
```

`msl.network.cli_keygen.`**`add_parser_keygen`**`(`*parser*`)`

Add the `keygen` command to the *parser*.

`msl.network.cli_keygen.`**`execute`**`(`*args*`)`

Executes the `keygen` command.

1.9.6 msl.network.cli_start module

Command line interface for the `start` command.

To see the help documentation, run the following command in a terminal:

```
msl-network start --help
```

Start the MSL Network Manager.

Examples:

```
# start the Network Manager using the default settings
msl-network start

# start the Network Manager on port 8326
msl-network start --port 8326

# require an authentication password for Clients and Services
# to be able to connect to the Network Manager
```

(continues on next page)

(continued from previous page)

```
msl-network start --auth-password abc 123

# use a specific certificate and key for the secure TLS protocol
msl-network start --cert-file /path/to/cert.pem --key-file /path/to/key.pem

# require that a valid username and password are specified for
# Clients and Services to be able to connect to the Network Manager
msl-network start --auth-login
```

See Also:

```
msl-network certgen
msl-network keygen
msl-network hostname
msl-network user
```

`msl.network.cli_start.`**`add_parser_start`**`(`*parser*`)`

Add the `start` command to the *parser*.

`msl.network.cli_start.`**`execute`**`(`*args*`)`

Executes the `start` command.

1.9.7 msl.network.cli_user module

Command line interface for the `user` command.

To see the help documentation, run the following command in a terminal:

```
msl-network user --help
```

Add/remove a user into/from a database.

The Network Manager can be started with the option to use a user's login credentials as the authorisation check for a Client or Service to be able to connect to the Network Manager.

To use the login credentials as the authentication check, start the Network Manager with the `--auth-login` flag:

```
msl-network start --auth-login
```

Examples:

```
# add 'j.doe' to the default database
msl-network user add j.doe --password a good password

# add 'a.smith' as an administrator to the database
msl-network user add a.smith --password !PaSsWoRd* --admin

# update 'j.doe' to be an administrator
msl-network user update j.doe --admin

```

(continues on next page)

(continued from previous page)

```
# update 'a.smith' to not be an administrator
msl-network user update a.smith

# update the password for 'j.doe' using a password in a file
msl-network user update j.doe --password /path/to/my/password.txt

# remove 'j.doe' from the default database
msl-network user remove j.doe

# add 'j.doe' to a specific database
msl-network user add j.doe --password The Password To Use --database /path/to/
↪database.db

# list all users in the database
msl-network user list
```

`msl.network.cli_user.`**`add_parser_user`**(*parser*)

Add the `user` command to the *parser*.

`msl.network.cli_user.`**`execute`**(*args*)

Executes the `user` command.

For example, run

```
msl-network start --help
```

from a terminal to print the help for the `start` command.

1.10 MSL-Network API Documentation

MSL-Network has very little functions or classes that need to be accessed in a user's application.

Typically, only the `Service` class needs to be subclassed and the `connect()` function will be called to connect to the Network `Manager` for most applications using MSL-Network.

The `msl.network.ssh` module provides some functions for using SSH to connect to a remote computer. *Starting a Service from another computer* shows an example Python package that can automatically start a Network `Manager` and a `Service` on a Raspberry Pi from another computer.

The process of establishing a connection to a `Manager` and linking with a particular `Service` can be achieved by creating a `LinkedClient`. This can be useful if you only want to link with a single `Service` on a `Manager`.

1.10.1 Package Structure

msl.network package

Concurrent and asynchronous network I/O.

`msl.network.`**`version_info = version_info(major=1, minor=0, micro=0, releaselevel='final')`**

Contains the version information as a (major, minor, micro, releaselevel) tuple.

Type

`namedtuple`

msl.network.client module

Use the `connect()` function to connect to a Network `Manager` as a `Client`.

`msl.network.client.`**`connect`**(**, name='Client', host='localhost', port=1875, timeout=10, username=None, password=None, password_manager=None, read_limit=None, disable_tls=False, cert_file=None, assert_hostname=True, auto_save=False*)

Create a new connection to a Network `Manager` as a `Client`.

Changed in version 0.4: Renamed *certificate* to *certfile*.

Changed in version 1.0: Renamed *certfile* to *cert_file*. Added the *auto_save* and *read_limit* keyword arguments.

Parameters

- **`name`** (`str`, optional) – A name to assign to the `Client` to help identify it on the network.
- **`host`** (`str`, optional) – The hostname (or IP address) of the Network `Manager` that the `Client` should connect to.
- **`port`** (`int`, optional) – The port number of the Network `Manager` that the `Client` should connect to.
- **`timeout`** (`float`, optional) – The maximum number of seconds to wait to connect to the Network `Manager`.
- **`username`** (`str`, optional) – The username to use to connect to the Network `Manager`. You need to specify a username to connect to a `Manager` only if the `Manager` was started using the `--auth-login` flag. If a username is required, and you have not specified a value then you will be asked for a username. See `cli_start` for more details.
- **`password`** (`str`, optional) – The password that is associated with *username*. If a password is required, and you have not specified a value then you will be asked for the password.
- **`password_manager`** (`str`, optional) – The password that is associated with the Network `Manager`. You need to specify the password only if the Network `Manager` was started using the `--auth-password` flag. If a password is required, and you have not specified a value then you will be asked for the password.

- **read_limit** (`int`, optional) – The buffer size limit when reading bytes from a network stream. If `None` then there is no (practical) limit.
- **disable_tls** (`bool`, optional) – Whether to connect to the Network *`Manager`* with or without using the secure TLS protocol.
- **cert_file** (`str`, optional) – The path to a certificate file to use for the secure TLS connection with the Network *`Manager`*. Not used if *disable_tls* is `True`.
- **assert_hostname** (`bool`, optional) – Whether to check that the hostname of the Network *`Manager`* matches the value of *host*. Not used if *disable_tls* is `True`.
- **auto_save** (`bool`, optional) – Whether to automatically save the certificate of the Network *`Manager`* if the certificate is not already saved. Not used if *disable_tls* is `True`.

Returns

`Client` – A new connection to a Network *`Manager`*.

`msl.network.client.`**`filter_client_connect_kwargs`**`(`*****kwargs***`)`

From the specified keyword arguments only return those that are valid for *`connect()`*.

New in version 0.4.

Parameters

kwargs – All keyword arguments that are not in the function signature of *`connect()`* are silently ignored and are not included in the output.

Returns

`dict` – Valid keyword arguments that can be passed to *`connect()`*.

class `msl.network.client.`**`Client`**`(`*name*`)`

Bases: *`Device`*

Base class for all Clients.

Attention: Do not instantiate directly. Use *`connect()`* to connect to a Network *`Manager`*.

`admin_request``(`*attrib*, **args*, ***kwargs*`)`

Send a request to the Network *`Manager`* as an administrator.

The user that calls this method must have administrative privileges for that *`Manager`*. See *`cli_user`* for details on how to create a user that is an administrator .

Changed in version 0.3: Added a *timeout* option as one of the keyword arguments.

Parameters

- **attrib** (`str`) – The attribute of the *`Manager`*. Can contain dots **.** to access sub-attributes.
- ***args** – The arguments to send to *attrib* of the *`Manager`*.
- ****kwargs** – The keyword arguments to send to *attrib* of the *`Manager`*. Also accepts a *timeout* keyword argument as a `float` or `int` as the maximum number of seconds to wait for the reply from the Network *`Manager`*. The default timeout is `None`.

Returns
The reply from the Network *Manager*.

Examples

```
>>> from msl.network import connect
>>> cxn = connect(**kwargs)
>>> cxn.admin_request('users_table.usernames')
['Alice', 'Bob', 'Charlie', 'Eve', 'admin']
>>> cxn.admin_request('users_table.is_user_registered', 'N.Bohr')
False
```

An admin can also shut down the *Manager*

```
>>> from msl.network.constants import SHUTDOWN_MANAGER
>>> cxn.admin_request(SHUTDOWN_MANAGER)
```

disconnect(*timeout=None*)

Disconnect from the Network *Manager*.

Changed in version 1.0: Added the *timeout* keyword argument.

Parameters
timeout (int or float, optional) – The maximum number of seconds to wait for the reply from the Network *Manager*.

is_connected()

Whether the *Client* is currently connected to the Network *Manager*.

New in version 1.0.

Returns
bool – Whether the connection is active.

link(*service*, ***, *timeout=None*)

Link with a *Service* on the Network *Manager*.

Changed in version 0.3: Added the *timeout* keyword argument.

Parameters

- **service** (str) – The name of the *Service* to link with.
- **timeout** (int or float, optional) – The maximum number of seconds to wait for the reply from the Network *Manager*.

Returns
Link – A *Link* with the requested *service*.

identities(***, *as_string=False*, *indent=2*, *timeout=None*)

Returns the identities of all devices that are connected to the Network *Manager*.

Changed in version 0.3: Added the *timeout* keyword argument.

Changed in version 0.4: Renamed *as_yaml* to *as_string*.

Changed in version 1.0: Renamed this method from *manager* to *identities*.

Parameters

- **as_string** (`bool`, optional) – Whether to return the information from the Network *`Manager`* as a *human-readable* string.
- **indent** (`int`, optional) – The amount of indentation added for each recursive level. Only used if *as_string* is `True`.
- **timeout** (`int` or `float`, optional) – The maximum number of seconds to wait for the reply from the Network *`Manager`*.

Returns

`dict` or `str` – The identities of all connected devices.

spawn(*name='Client'*)

Returns a new connection to the Network *`Manager`*.

Parameters

name (`str`, optional) – The name to assign to the new *`Client`*.

Returns

`Client` – A new Client.

unlink(*link*, ***, *timeout=None*)

Unlink from a *`Service`* on the Network *`Manager`*.

New in version 0.5.

Parameters

- **link** (*`Link`*) – The object that is linked with the *`Service`*.
- **timeout** (`int` or `float`, optional) – The maximum number of seconds to wait for the reply from the Network *`Manager`*.

class msl.network.client.**Link**(*client*, *service*, *identity*)

Bases: `object`

A network link between a *`Client`* and a *`Service`*.

Attention: Not to be instantiated directly. A *`Client`* creates a *`Link`* via the *`Client.link()`* method.

acquire_lock(*shared=False*, *timeout=None*)

Acquire a lock with the linked *`Service`*.

When a lock is acquired, no more *`Client`*s are allowed to link with the *`Service`* until all locks have been released.

If *`service_max_clients`* returns a value of 1, then there is no need to acquire a lock since only a single *`Client`* can link with the *`Service`* at a time.

New in version 1.0.

Parameters

- **shared** (`bool`, optional) – Whether the lock is exclusive or shared. An exclusive lock can only be acquired if a single *`Client`* is linked with the *`Service`*. A shared lock allows for multiple simultaneous links, however, once any of the linked `Client`s requests a lock the lock is shared amongst the currently-linked `Client`s and no new `Client`s can link with the `Service` until all locks have been released.

- **`timeout`** (`int` or `float`, optional) – The maximum number of seconds to wait for the reply from the Network `Manager`.

Returns
`list` of `str` – The names of the `Client`s that are linked with the `Service` while the lock is active. For an exclusive lock, only a single link is allowed so the list contains a single item that is the name of the `Client` that requested the lock.

Raises
`RuntimeError` – If a lock cannot be acquired.

`release_lock`(*timeout=None*)

Release a lock with the linked `Service`.

New in version 1.0.

Parameters
`timeout` (`int` or `float`, optional) – The maximum number of seconds to wait for the reply from the Network `Manager`.

Returns
`list` of `str` – The names of the `Client`s that still have a lock with the `Service` after this lock has been released. An emtpy list means that there are no active locks.

`property service_address`

The address of the `Service` that this object is linked with.

Type
`str`

`property service_attributes`

The attributes of the `Service` that this object is linked with.

Type
`dict`

`property service_language`

The programming language that the `Service` is running on.

Type
`str`

`property service_max_clients`

The maximum number of `Client`s that can be linked with the `Service`. A value ≤ 0 means that there is no limit.

New in version 1.0.

Type
`int`

property service_name

The name of the *Service* that this object is linked with.

Type

str

property service_os

The operating system that the *Service* is running on.

Type

str

disconnect(*timeout=None*)

An alias for *unlink()*.

New in version 0.5.

notification_handler(**args*, ***kwargs*)

Handle a notification from the *Service* that emitted a notification.

Important: You must re-assign this method at the instance level in order to handle the notification.

New in version 0.5.

Parameters

- **args** – The arguments that were emitted.
- **kwargs** – The keyword arguments that were emitted.

Examples

The following assumes that the *Heartbeat Service* is running on the same computer. Using types.MethodType allows for the *print_notification* function to access the *self* attribute of *heartbeat*.

```
>>> import types
>>> from msl.network import connect
>>> cxn = connect()
>>> heartbeat = cxn.link('Heartbeat')
>>> def print_notification(self, *args, **kwargs):
...     print(f'The {self.service_name} Service emitted', args,␣
↪kwargs)
...
>>> heartbeat.notification_handler = types.MethodType(print_
↪notification, heartbeat)
The Heartbeat Service emitted (72,) {}
The Heartbeat Service emitted (73,) {}
The Heartbeat Service emitted (74,) {}
The Heartbeat Service emitted (75,) {}
The Heartbeat Service emitted (76,) {}
The Heartbeat Service emitted (77,) {}
```

(continues on next page)

(continued from previous page)

```
>>> heartbeat.reset()
The Heartbeat Service emitted (0,) {}
The Heartbeat Service emitted (1,) {}
The Heartbeat Service emitted (2,) {}
The Heartbeat Service emitted (3,) {}
The Heartbeat Service emitted (4,) {}
The Heartbeat Service emitted (5,) {}
The Heartbeat Service emitted (6,) {}
>>> heartbeat.kill()
>>> cxn.disconnect()
```

See also:

emit_notification(), *emit_notification_threadsafe()*

shutdown_service(**args*, ***kwargs*)

Send a request for the *Service* to shut down.

A *Service* must also implement a method called `shutdown_service` otherwise calling this *shutdown_service()* method will raise an exception.

See *Starting a Service from another computer* for an example use case.

New in version 0.5.

Parameters

- **args** – The positional arguments that are passed to the `shutdown_service` method of the *Service* that this object is linked with.
- **kwargs** – The keyword arguments that are passed to the `shutdown_service` method of the *Service* that this object is linked with. Also accepts a *timeout* keyword argument as a `float` or `int` as the maximum number of seconds to wait for the reply from the Network *Manager*. The default timeout is `None`.

Returns

Whatever the `shutdown_service` method of the *Service* returns.

unlink(*timeout=None*)

Unlink from the *Service* on the Network *Manager*.

New in version 0.5.

Parameters

timeout (`int` or `float`, optional) – The maximum number of seconds to wait for the reply from the Network *Manager*.

class msl.network.client.**LinkedClient**(*service_name*, ***kwargs*)

Bases: `object`

Create a new *Client* that has a *Link* with the specified *Service*.

New in version 0.4.

Parameters

- **service_name** (`str`) – The name of the *Service* to *link* with.

- **kwargs** – Keyword arguments that are passed to `connect()`.

acquire_lock(*shared=False*, *timeout=None*)

See `Link.acquire_lock` for more details.

admin_request(*attrib*, **args*, ***kwargs*)

See `Client.admin_request` for more details.

disconnect(*timeout=None*)

See `Client.disconnect` for more details.

identity()

See `identity` for more details.

identities(***, *as_string=False*, *indent=2*, *timeout=None*)

See `Client.identities` for more details.

is_connected()

See `Client.is_connected` for more details.

notification_handler(**args*, ***kwargs*)

See `Link.notification_handler` for more details.

service_error_handler()

This method is called immediately before an exception is raised if there was an error processing a request on the `Service` that this object is linked with.

You can override this method to perform any necessary cleanup (e.g., closing file handles, shutting down threads, disconnecting from devices, etc.) before a `RuntimeError` is raised.

The `Service` remains running. This is to clean up the `Client` instance.

shutdown_service(**args*, ***kwargs*)

See `Link.shutdown_service` for more details.

spawn(*name='LinkedClient'*)

Returns a new connection to the Network `Manager` that has a `Link` with the same `Service`.

Parameters
name (`str`, optional) – The name to assign to the new `Client`.

Returns
`LinkedClient` – A new `Client` that has a `Link` with the same `Service`.

unlink(*timeout=None*)

See `Link.unlink` for more details.

property address_manager

See `address_manager` for more details.

property client

The `Client` that is providing the `Link`.

New in version 0.5.

Type
`Client`

property link

The *Link* with the *Service*.

Type

Link

property name

See *name* for more details.

property port

See *port* for more details.

release_lock(*timeout=None*)

See *Link.release_lock* for more details.

property service_address

See *Link.service_address* for more details.

property service_attributes

See *Link.service_attributes* for more details.

property service_language

See *Link.service_language* for more details.

property service_max_clients

See *Link.service_max_clients* for more details.

property service_name

See *Link.service_name* for more details.

property service_os

See *Link.service_os* for more details.

msl.network.constants module

Constants that are used by the MSL-Network package.

`msl.network.constants.`**`PORT = 1875`**

The default port number to use for the Network *Manager* (the year that the BIPM was established).

Type

int

`msl.network.constants.`**`HOSTNAME = 'build-21031035-project-167229-msl-network'`**

The hostname of the computer.

Type

str

`msl.network.constants.`**`HOME_DIR = '/home/docs/.msl/network'`**

The default directory where all files are to be located.

Can be overwritten by specifying a `MSL_NETWORK_HOME` environment variable.

Type

str

`msl.network.constants.`**`CERT_DIR = '/home/docs/.msl/network/certs'`**

The default directory to save PEM certificates.

Type
`str`

`msl.network.constants.`**`KEY_DIR = '/home/docs/.msl/network/keys'`**

The default directory to save private PEM keys.

Type
`str`

`msl.network.constants.`**`DATABASE = '/home/docs/.msl/network/manager.sqlite3'`**

The default database path.

Type
`str`

`msl.network.constants.`**`IS_WINDOWS = False`**

Whether the operating system is Windows.

Type
`bool`

`msl.network.constants.`**`IS_LINUX = True`**

Whether the operating system is Linux.

Type
`bool`

`msl.network.constants.`**`LOCALHOST_ALIASES = {'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.ip6.arpa', '1.0.0.127.in-addr.arpa', '127.0.0.1', '::1', 'build-21031035-project-167229-msl-network', 'localhost'}`**

Aliases for `localhost`.

Type
`set` of `str`

msl.network.cryptography module

Functions to create a self-signed certificate for the secure SSL/TLS protocol.

`msl.network.cryptography.`**`generate_key`**`(`**, path=None, algorithm='RSA', password=None, size=2048, curve='SECP384R1'*`)`

Generate a new private key.

Parameters

- **`path`** (`str`, optional) – The path to save the private key to. If not specified then save the private key in the default directory with the default filename.
- **`algorithm`** (`str`, optional) – The encryption algorithm to use to generate the private key. Options are:
  - `RSA` - Rivest, Shamir, and Adleman algorithm.
  - `DSA` - Digital Signature Algorithm.

  - ECC - Elliptic Curve Cryptography.
- **password** (str, optional) – The password to use to encrypt the key.
- **size** (int, optional) – The size (number of bits) of the key. Only used if *algorithm* is RSA or DSA.
- **curve** (str, optional) – The name of the elliptic curve to use. Only used if *algorithm* is ECC. See Elliptic Curves for example elliptic-curve names.

Returns

str – The path to the private key.

`msl.network.cryptography.`**`load_key`**`(`*path*, *, *password=None*`)`

Load a private key from a file.

Parameters

- **path** (str) – The path to a key file.
- **password** (str, optional) – The password to use to decrypt the private key.

Returns

RSAPrivateKey, DSAPrivateKey or EllipticCurvePrivateKey – The private key.

`msl.network.cryptography.`**`generate_certificate`**`(`*, *path=None*, *key_path=None*, *key_password=None*, *algorithm='SHA256'*, *years_valid=None*, *digest_size=None*, *name=None*, *extensions=None*`)`

Generate a self-signed certificate.

Changed in version 1.0: Added the *digest_size*, *name* and *extensions* keyword arguments.

Parameters

- **path** (str, optional) – The path to save the certificate to. If not specified then save the certificate in the default directory with the default filename.
- **key_path** (str, optional) – The path to the private key which will be used to digitally sign the certificate. If not specified then automatically generates a new private key (overwriting the default private key if one already exists).
- **key_password** (str, optional) – The password to use to decrypt the private key.
- **algorithm** (str or HashAlgorithm, optional) – The hash algorithm to use. See Message digests (Hashing) for allowed hash algorithms.
- **years_valid** (int or float, optional) – The number of years that the certificate is valid for. If you want to specify that the certificate is valid for 3 months then set *years_valid* to be 0.25. Default is 100 years for 64-bit platforms and 15 years for 32-bit platforms.
- **digest_size** (int, optional) – The digest size (if the hash *algorithm* requires one).
- **name** (Name, optional) – The object to use for the `subject_name()` and the `issuer_name()`. If not specified then a default *name* is used.

- **extensions** (list of ExtensionType, optional) – The extensions to add to the certificate.

Returns
str – The path to the self-signed certificate that was generated.

msl.network.cryptography.**load_certificate**(*cert*)

Load a PEM certificate.

Parameters
cert (str or bytes) – If str then the path to the certificate file. If bytes then the raw certificate data.

Returns
Certificate – The PEM certificate.

Raises
TypeError – If *cert* is not of type str or bytes.

msl.network.cryptography.**get_default_cert_path**()

str: Returns the default certificate path.

msl.network.cryptography.**get_default_key_path**()

str: Returns the default key path.

msl.network.cryptography.**get_fingerprint**(*cert*, ***, *algorithm='SHA1'*, *digest_size=None*)

Get the fingerprint of a certificate.

Changed in version 1.0: Added the *digest_size* keyword argument and allow *algorithm* to be a string.

Parameters

- **cert** (Certificate) – The PEM certificate.
- **algorithm** (str or HashAlgorithm, optional) – The hash algorithm to use. See Message digests (Hashing) for allowed hash algorithms.
- **digest_size** (int, optional) – The digest size (if the hash *algorithm* requires one).

Returns
str – The fingerprint as a colon-separated hex string.

msl.network.cryptography.**get_metadata**(*cert*)

Get the metadata of a certificate.

Parameters
cert (Certificate) – The certificate.

Returns
dict – The metadata of the certificate.

msl.network.cryptography.**get_metadata_as_string**(*cert*)

Returns the metadata of a certificate as a *human-readable* string.

Parameters
cert (Certificate) – The certificate.

Returns
`str` – The metadata of the certificate.

`msl.network.cryptography.`**`get_ssl_context`**`(`*, *cert_file=None*, *host=None*, *port=None*, *auto_save=False*, ***kwargs*`)`

Get the SSL context.

Gets the context either from connecting to a remote server or from loading it from a file.

To get the context from a remote server you must specify both *host* and *port*.

Changed in version 0.4: Renamed *certificate* to *certfile*.

Changed in version 1.0: Renamed *certfile* to *cert_file*. Added the *auto_save* keyword argument and ***kwargs*.

Parameters

- **`cert_file`** (`str`, optional) – The path to a certificate file to load. If specified then *host*, *port* and *auto_save* are ignored.
- **`host`** (`str`, optional) – The hostname or IP address of the remote server to connect to.
- **`port`** (`int`, optional) – The port number of the remote server to connect to.
- **`auto_save`** (`bool`, optional) – Whether to automatically save the certificate from the server. Default is to ask before saving.
- **`**kwargs`** – All additional keyword arguments are passed to `ssl.get_server_certificate()`.

Returns

- `str` – The path to the certificate file that was loaded.
- `ssl.SSLContext` – The SSL context.

msl.network.database module

Databases that are used by the Network *`Manager`*.

`class` `msl.network.database.`**`Database`**`(`*database*, ***kwargs*`)`

Bases: `object`

Base class for connecting to a SQLite database.

Automatically creates the database if it does not already exist.

Parameters

- **`database`** (`str`) – The path to the database file, or `':memory:'` to open a connection to a database that resides in RAM instead of on disk.
- **`kwargs`** – Optional keyword arguments to pass to `sqlite3.connect()`.

`property path`

The path to the database file.

Type
`str`

property connection

The connection object.

Type
`sqlite3.Connection`

property cursor

The cursor object.

Type
`sqlite3.Cursor`

close()

Closes the connection to the database.

execute(*sql*, *parameters=None*)

Wrapper around `sqlite3.Cursor.execute()`.

Parameters

- **sql** (`str`) – The SQL command to execute
- **parameters** (`list`, `tuple` or `dict`, optional) – Only required if the *sql* command is parameterized.

tables()

`list` of `str`: A list of the names of each table that is in the database.

table_info(*name*)

Returns the information about each column in the specified table.

Parameters
name (`str`) – The name of the table to get the information of.

Returns

`list` of `tuple` – The list of the fields in the table. The indices of each tuple correspond to:

- 0 - id number of the column
- 1 - the name of the column
- 2 - the datatype of the column
- 3 - whether a value in the column can be NULL (0 or 1)
- 4 - the default value for the column
- 5 - whether the column is used as a primary key (0 or 1)

column_names(*table_name*)

Returns the names of the columns in the specified table.

Parameters
table_name (`str`) – The name of the table.

Returns
`list` of `str` – A list of the names of each column in the table.

`column_datatypes`(*table_name*)

Returns the datatype of each column in the specified table.

Parameters

`table_name` (`str`) – The name of the table.

Returns

`list` of `str` – A list of the datatypes of each column in the table.

class `msl.network.database.`**`ConnectionsTable`**(*, *database=None*, *as_datetime=False*, ***kwargs*)

Bases: *`Database`*

The database table for devices that have connected to the Network *`Manager`*.

Parameters

- **`database`** (`str`, optional) – The path to the database file, or `':memory:'` to open a connection to a database that resides in RAM instead of on disk. If `None` then loads the default database.
- **`as_datetime`** (`bool`, optional) – Whether to fetch the timestamps from the database as `datetime.datetime` objects. If `False` then the timestamps will be of type `str`.
- **`kwargs`** – Optional keyword arguments to pass to `sqlite3.connect()`.

`NAME = 'connections'`

The name of the table in the database.

Type

`str`

`insert`(*peer*, *message*)

Insert a message about what happened when a device connected.

Parameters

- **`peer`** (*`Peer`*) – The peer that connected to the Network *`Manager`*.
- **`message`** (`str`) – The message about what happened (e.g, the connection was successful, or it failed).

`connections`(*, *start=None*, *end=None*)

Return the information of the devices that have connected to the Network *`Manager`*.

Changed in version 1.0: Use `T` as the separator between the date and time. Renamed *timestamp1* to *start*. Renamed *timestamp2* to *end*.

Parameters

- **`start`** (`datetime.datetime` or `str`, optional) – Include all records that have a timestamp $\geq$ *start*. If a `str` then in the `yyyy-mm-dd` or `yyyy-mm-ddTHH:MM:SS` format.
- **`end`** (`datetime.datetime` or `str`, optional) – Include all records that have a timestamp $\leq$ *end*. If a `str` then in the `yyyy-mm-dd` or `yyyy-mm-ddTHH:MM:SS` format.

Returns
`list` of `tuple` – The connection records.

class `msl.network.database.`**`HostnamesTable`**(*, *database=None*, ***kwargs*)

Bases: *`Database`*

The database table for trusted hostname's that are allowed to connect to the Network *`Manager`*.

Parameters

- **`database`** (`str`, optional) – The path to the database file, or `':memory:'` to open a connection to a database that resides in RAM instead of on disk. If `None` then loads the default database.
- **`kwargs`** – Optional keyword arguments to pass to `sqlite3.connect()`.

`NAME = 'auth_hostnames'`

The name of the table in the database.

Type
`str`

`insert`(*hostname*)

Insert a hostname.

If the hostname is already in the table then it does not insert it again.

Parameters
`hostname` (`str`) – The trusted hostname.

`delete`(*hostname*)

Delete a hostname.

Parameters
`hostname` (`str`) – A hostname in the table.

Raises
`ValueError` – If *hostname* is not in the table.

`hostnames`()

`list` of `str`: Returns all the trusted hostnames.

class `msl.network.database.`**`UsersTable`**(*, *database=None*, ***kwargs*)

Bases: *`Database`*

The database table for keeping information about a users login credentials for connecting to a Network *`Manager`*.

Parameters

- **`database`** (`str`, optional) – The path to the database file, or `':memory:'` to open a connection to a database that resides in RAM instead of on disk. If `None` then loads the default database.
- **`kwargs`** – Optional keyword arguments to pass to `sqlite3.connect()`.

`NAME = 'auth_users'`

The name of the table in the database.

Type
`str`

insert(*username*, *password*, *is_admin*)

Insert a new user.

The password is encrypted and stored in the database using PBKDF2

To update the values for a user use `update()`.

Parameters

- **username** (`str`) – The name of the user.
- **password** (`str`) – The password of the user in plain-text format.
- **is_admin** (`bool`) – Does this user have admin rights?

Raises
`ValueError` – If the *username* is invalid or if *password* is empty.

update(*username*, *, *password=None*, *is_admin=None*)

Update either the salt used for the password and/or the admin rights.

Parameters

- **username** (`str`) – The name of the user.
- **password** (`str`, optional) – The password of the user in plain-text format.
- **is_admin** (`bool`, optional) – Does this user have admin rights?

Raises
`ValueError` – If *username* is not in the table. If both *password* and *is_admin* are not specified. If *password* is an empty string.

delete(*username*)

Delete a user.

Parameters
username (`str`) – The name of the user.

Raises
`ValueError` – If *username* is not in the table.

get_user(*username*)

Get the information about a user.

Parameters
username (`str`) – The name of the user.

Returns
`tuple` – Returns (pid, username, key, salt, is_admin) for the specified *username*.

records()

`list` of `tuple`: Returns [(pid, username, key, salt, is_admin), . . .] for all users.

usernames()

`list` of `str`: Returns the names of all registered users.

users()

list of tuple: Returns [(username, is_admin), …] for all users.

is_user_registered(*username*)

bool: Whether *username* is a registered user.

is_password_valid(*username*, *password*)

Check whether the password matches the encrypted password in the database.

Parameters

- **username** (str) – The name of the user.
- **password** (str) – The password to check (in plain-text format).

Returns

bool – Whether *password* matches the password in the database for the user.

is_admin(*username*)

Check whether a user has admin rights.

Parameters

username (str) – The name of the user.

Returns

bool – Whether the user has admin rights.

msl.network.database.**convert_datetime**(*value*)

Convert a date and time to a datetime object.

Parameters

value (bytes) – The datetime value from an SQLite database.

Returns

datetime.datetime – The *value* as a datetime object.

msl.network.json module

This module is used as the JSON (de)serializer.

class msl.network.json.**Package**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Supported Python packages for (de)serializing JSON objects.

By default, the builtin `json` module is used.

To change which JSON package to use you can call *use()* to set the backend during runtime, or you can specify an `MSL_NETWORK_JSON` environment variable as the default backend. For example, creating an environment variable named `MSL_NETWORK_JSON` and setting its value to be `ULTRA` would use UltraJSON to (de)serialize JSON objects.

Changed in version 1.0: Moved from the *msl.network.constants* module and renamed. Added `JSON`, `UJSON`, `RAPIDJSON` and `SIMPLEJSON` aliases. Added `OR` (and alias `ORJSON`) for orjson. Removed `YAJL`.

BUILTIN = 'BUILTIN'

json

JSON = 'BUILTIN'

json

ULTRA = 'ULTRA'

UltraJSON

UJSON = 'ULTRA'

UltraJSON

RAPID = 'RAPID'

RapidJSON

RAPIDJSON = 'RAPID'

RapidJSON

SIMPLE = 'SIMPLE'

simplejson

SIMPLEJSON = 'SIMPLE'

simplejson

OR = 'OR'

orjson

ORJSON = 'OR'

orjson

`msl.network.json.`**`use`**`(`*value*`)`

Set which JSON backend to use.

New in version 1.0.

Parameters

value (*Package* or str) – An enum value or member name (case-insensitive).

Examples

```
>>> from msl.network import json
>>> json.use(json.Package.UJSON)
>>> json.use('ujson')
```

`msl.network.json.`**`serialize`**`(`*obj*`)`

Serialize an object as a JSON-formatted string.

Parameters

obj – A JSON-serializable object.

Returns

str – The JSON-formatted string.

`msl.network.json.`**`deserialize`**`(`*s*`)`

Deserialize a JSON-formatted string to Python objects.

Parameters
s (`str`, `bytes` or `bytearray`) – A JSON-formatted string.

Returns
The deserialized Python object.

msl.network.manager module

The Network *`Manager`*.

`class` `msl.network.manager.`**`Manager`**`(`*port*, *password*, *login*, *hostnames*, *connections_table*, *users_table*, *hostnames_table*, *loop*`)`

Bases: *`Network`*

The Network *`Manager`*.

Attention: Not to be instantiated directly. Start the Network *`Manager`* from the command line. Run `msl-network start --help` from a terminal for more information.

`async acquire_lock``(`*writer*, *uid*, *service*, *shared*`)`

A request from a *`Client`* to lock a *`Service`*.

New in version 1.0.

Parameters

- **`writer`** (`asyncio.StreamWriter`) – The stream writer of the *`Client`*.
- **`uid`** (`str`) – The unique identifier of the request.
- **`service`** (`str`) – The name of the *`Service`* that the *`Client`* wants to acquire a lock with.
- **`shared`** (`bool`) – Whether the lock is exclusive or shared.

`async new_connection``(`*reader*, *writer*`)`

Receive a new connection request.

To accept the new connection request, the following checks must be successful:

1. The correct authentication reply is received.
2. A correct *`identity`* is received, i.e., is the connection from a *`Client`* or *`Service`*?

Parameters

- **`reader`** (`asyncio.StreamReader`) – The stream reader.
- **`writer`** (`asyncio.StreamWriter`) – The stream writer.

`async check_user``(`*reader*, *writer*`)`

Check the login credentials of a user.

Parameters

- **reader** (asyncio.StreamReader) – The stream reader.
- **writer** (asyncio.StreamWriter) – The stream writer.

Returns
bool – Whether the login credentials are valid.

async check_manager_password(*reader*, *writer*)

Check the *Manager*'s password from the connected device.

Parameters

- **reader** (asyncio.StreamReader) – The stream reader.
- **writer** (asyncio.StreamWriter) – The stream writer.

Returns
bool – Whether the correct password was received.

async check_identity(*reader*, *writer*)

Check the *identity* of the connected device.

Parameters

- **reader** (asyncio.StreamReader) – The stream reader.
- **writer** (asyncio.StreamWriter) – The stream writer.

Returns
str or None – If the identity check was successful then returns the connection type, either 'client' or 'service', otherwise returns None.

async get_handshake_data(*reader*)

Used by *check_manager_password()*, *check_identity()* and *check_user()*.

Parameters
reader (asyncio.StreamReader) – The stream reader.

Returns
None, str or dict – The data.

async handler(*reader*, *writer*)

Handles requests from the connected *Client*s and replies or notifications from the connected *Service*s.

Parameters

- **reader** (asyncio.StreamReader) – The stream reader.
- **writer** (asyncio.StreamWriter) – The stream writer.

async release_lock(*writer*, *uid*, *service*)

A request from a *Client* to unlock a *Service*.

New in version 1.0.

Parameters

- **writer** (asyncio.StreamWriter) – The stream writer of the *Client*.
- **uid** (str) – The unique identifier of the request.

- **service** (`str`) – The name of the *Service* that the *Client* wants to release a lock with.

`async remove_peer(id_type, writer)`

Remove this peer from the registry of connected peers.

Parameters

- **id_type** (`str`) – The type of the connection, either `'client'` or `'service'`.
- **writer** (`asyncio.StreamWriter`) – The stream writer of the peer.

`async close_writer(writer)`

Close the connection to the `asyncio.StreamWriter`.

Log that the connection is closing, drains the writer and then closes the connection.

Parameters

writer (`asyncio.StreamWriter`) – The stream writer to close.

`async shutdown_manager()`

Disconnect all *Service*s and *Client*s from the *Manager* and then shut down the *Manager*.

`identity()`

`dict`: The *identity* of the Network *Manager*.

`async link(writer, uid, service)`

A request from a *Client* to link it with a *Service*.

Parameters

- **writer** (`asyncio.StreamWriter`) – The stream writer of the *Client*.
- **uid** (`str`) – The unique identifier of the request.
- **service** (`str`) – The name of the *Service* that the *Client* wants to link with.

`async unlink(writer, uid, service)`

A request from a *Client* to unlink it from a *Service*.

New in version 0.5.

Parameters

- **writer** (`asyncio.StreamWriter`) – The stream writer of the *Client*.
- **uid** (`str`) – The unique identifier of the request.
- **service** (`str`) – The name of the *Service* that the *Client* wants to unlink from.

`async write_request(writer, attribute, *args, **kwargs)`

Write a request to a *Client* or to a *Service*.

Parameters

- **writer** (`asyncio.StreamWriter`) – The stream writer of the *Client* or *Service*.
- **attribute** (`str`) – The name of the attribute to request.

- **args** – The arguments that *attribute* requires.
- **kwargs** – The key-value pairs that *attribute* requires.

`class msl.network.manager.Peer(writer)`

Bases: `object`

Metadata about a peer that is connected to the Network `Manager`.

Attention: Not to be called directly. To be called when the Network `Manager` receives a `new_connection()` request.

Parameters

writer (`asyncio.StreamWriter`) – The stream writer for the peer.

`msl.network.manager.run_forever(*, host=None, port=1875, auth_hostname=False, auth_login=False, auth_password=None, database=None, disable_tls=False, cert_file=None, key_file=None, key_file_password=None, log_level='INFO', log_file=None)`

Start the event loop for the Network `Manager`.

This is a blocking function. It will not return until the event loop of the `Manager` has stopped.

New in version 0.4.

Changed in version 1.0: Renamed *certfile* to *cert_file*. Renamed *keyfile* to *key_file*. Renamed *keyfile_password* to *key_file_password*. Renamed *logfile* to *log_file*. Removed the *debug* keyword argument. Added the *log_level* keyword argument. Added the *host* keyword argument.

Parameters

- **host** (`str`, optional) – The hostname or IP address to run the Network `Manager` on. If unspecified then all network interfaces are used.
- **port** (`int`, optional) – The port number to run the Network `Manager` on.
- **auth_hostname** (`bool`, optional) – If `True` then only connections from trusted hosts are allowed. If enabling *auth_hostname* then do not specify an *auth_password* and do not enable *auth_login*. Run `msl-network hostname --help` for more details.
- **auth_login** (`bool`, optional) – If `True` then checks a users login credentials (the username and password) before a `Client` or `Service` successfully connects. If enabling *auth_login* then do not specify an *auth_password* and do not enable *auth_hostname*. Run `msl-network user --help` for more details.
- **auth_password** (`str`, optional) – The password of the Network `Manager`. Essentially, this can be a thought of as a single password that all `Client`s and `Service`s need to specify before the connection to the Network `Manager` is successful. Can be a path to a file that contains the password on the first line in the file (**WARNING!!** if the path does not exist then the value of the path becomes the password). If using an *auth_password* then do not enable *auth_login* nor *auth_hostname*.

- **database** (str, optional) – The path to the sqlite3 database that contains the records for the following tables – *ConnectionsTable*, *HostnamesTable*, *UsersTable*. If None then loads the default database.
- **disable_tls** (bool, optional) – Whether to use TLS for the communication protocol.
- **cert_file** (str, optional) – The path to the TLS certificate file. See load_cert_chain() for more details. Only required if using TLS.
- **key_file** (str, optional) – The path to the TLS key file. See load_cert_chain() for more details.
- **key_file_password** (str, optional) – The password to decrypt the *key_file*. See load_cert_chain() for more details. Can be a path to a file that contains the password on the first line in the file (**WARNING!!** if the path does not exist then the value of the path becomes the password).
- **log_level** (str or int, optional) – The logging level to initially use. Can also be changed via an *admin_request()*.
- **log_file** (str, optional) – The file path to write logging messages to. If None then uses the default file path.

msl.network.manager.**run_services**(**services*, ***kwargs*)

This function starts the Network *Manager* and then starts the specified *Service*s.

This is a convenience function for running the Network *Manager* only when the specified *Service*s are all connected to the *Manager*. Once all *Service*s disconnect from the *Manager* then the *Manager* shuts down.

This is a blocking call. It will not return until the event loop of the *Manager* has stopped.

New in version 0.4.

Parameters

- **services** – The *Service*s to run on the *Manager*. Each *Service* must be instantiated but not started. This *run_services()* function will start each *Service*.
- **kwargs** – Keyword arguments are passed to *run_forever()* and to *start()*. The keyword arguments that are passed to *run_forever()* and *start()* that are not valid for that function are silently ignored.

Examples

If you want to allow a *Client* to be able to shut down a *Service* then implement a public `shutdown_service()` method on the *Service*. For example, the following `shutdownable_example.py` is a script that starts a Network *Manager* and two *Service*s

```python
# shutdownable_example.py

from msl.network import Service, run_services

class AddService(Service):
```

(continues on next page)

(continued from previous page)

```
    def add(self, a, b):
        return a + b

    def shutdown_service(self, *args, **kwargs):
        # do whatever you need to do before the AddService shuts down
        # return whatever you want
        return True

class SubtractService(Service):

    def subtract(self, a, b):
        return a - b

    def shutdown_service(self, *args, **kwargs):
        # do whatever you need to do before the SubtractService shuts␣
↪down
        # return whatever you want
        return 'Success!'

run_services(AddService(), SubtractService())
```

Then the *Client* script could be

```
from msl.network import connect

cxn = connect()
a = cxn.link('AddService')
s = cxn.link('SubtractService')
assert a.add(1, 2) == 3
assert s.subtract(1, 2) == -1
a.shutdown_service()
s.shutdown_service()
```

When both *Service*s have shut down then the Network *Manager* will also shut down and the *run_services()* function will no longer be blocking the execution of `shutdownable_example.py`.

`msl.network.manager.`**`filter_run_forever_kwargs`**`(**kwargs)`

From the specified keyword arguments only return those that are valid for *run_forever()*.

New in version 0.4.

Parameters
kwargs – All keyword arguments that are not part of the function signature for *run_forever()* are silently ignored and are not included in the output.

Returns
`dict` – Valid keyword arguments that can be passed to *run_forever()*.

msl.network.network module

Base classes for a *Manager*, *Service* and *Client*.

class msl.network.network.**Network**

Bases: `object`

Base class for the *Manager*, *Service* and *Client*.

identity() → dict

The identity of a device on the network.

All devices on the network must be able to identify themselves to any other device that is connected to the network. There are 3 possible types of network devices – a *Manager*, a *Service* and a *Client*. The member names and JSON datatype for each network device is described below.

- *Manager*

  hostname: string
  The name of the computer that the Network *Manager* is running on.

  port: integer
  The port number that the Network *Manager* is running on.

  attributes: object
  An object (a Python `dict`) of public attributes that the Network *Manager* provides. Users who are an administrator of the Network *Manager* can request private attributes, see *admin_request()*.

  language: string
  The programming language that the Network *Manager* is running on.

  os: string
  The name of the operating system that the Network *Manager* is running on.

  clients: object
  An object (a Python `dict`) of all *Client* devices that are currently connected to the Network *Manager*.

  services: object
  An object (a Python `dict`) of all *Service* devices that are currently connected to the Network *Manager*.

- *Service*

  type: string
  This must be equal to `'service'` (case-insensitive).

  name: string
  The name to associate with the *Service* (can contain spaces).

  attributes: object
  An object (a Python `dict`) of the attributes that the *Service* provides. The keys are the method names and the values are the method signatures (expressed as a string).

  The *attributes* get populated automatically when subclassing *Service*. If you are creating a *Service* in another programming language then you can use the following as an example for how to define an *attributes* object:

```
{
  "pi": "() -> float",
  "add_integers": "(x:int, y:int) -> int",
  "scalar_multiply": "(a:float, data:List[floats]) ->␣
↪List[floats]"
}
```

This *Service* would provide a method named `pi` that takes no inputs and returns a floating-point number, a method named `add_integers` that takes parameters named `x` and `y` as integer inputs and returns an integer, and a method named `scalar_multiply` that takes parameters named `a` as a floating-point number and `data` as an array of floating-point numbers as inputs and returns an array of floating-point numbers.

The key **must** be equal to the name of the method that the *Service* provides; however, the value (the method signature) is only used as a helpful guide to let a `Client` know what the method takes as inputs and what the method returns. How you express the method signature is up to you. The above example could also be expressed as:

```
{
  "pi": "() -> 3.1415926...",
  "add_integers": "(int32 x, int32 y) -> x+y",
  "scalar_multiply": "(double a, *double data) -> *double
↪"
}
```

language: string, optional
The programming language that the `Service` is running on.

os: string, optional
The name of the operating system that the `Service` is running on.

max_clients: integer, optional
The maximum number of `Client`s that can be linked with the `Service`. If the value is ≤ 0 then that means that an unlimited number of `Client`s can be linked *(this is the default setting if max_clients is not specified)*.

- `Client`

  type: string
  This must be equal to `'client'` (case-insensitive).

  name: string
  The name to associate with the `Client` (can contain spaces).

  language: string, optional
  The programming language that the `Client` is running on.

  os: string, optional
  The name of the operating system that the `Client` is running on.

Returns
`dict` – The identity of the network device.

static set_logging_level(*level: str | int*) → bool

Set the logging level.

Parameters

level (int or str) – The logging level of the msl.network logger.

Returns

bool – Whether setting the logging level was successful.

class msl.network.network.Device(*name=None*)

Bases: *Network*

Base class for a *Service* and *Client*.

New in version 1.0.

Parameters

name (str, optional) – The name of the device as it will appear on the Network *Manager*. If not specified then the class name is used.

property address_manager

The address of the *Manager* that this device is connected to.

Type

str

property loop_thread_id

Identifier of the thread running the event loop.

Returns None if the event loop is not running.

New in version 1.0.

property name

The name of the device on the *Manager*.

Type

str

property port

The port number of this device that is being used for the connection to the *Manager*.

Type

int

add_tasks(**coros_or_futures*)

Additional tasks to run in the event loop.

New in version 1.0.

Parameters

coros_or_futures – Coroutines or futures that will be passed to asyncio.gather() when the event loop runs.

shutdown_handler()

Called after the connection to the Network *Manager* has been lost but before the event loop stops.

Override this method to do any necessary cleanup.

New in version 1.0.

msl.network.service module

Base class for all Services.

class msl.network.service.**Service**(***, *name=None*, *max_clients=None*, *ignore_attributes=None*)

Bases: *Device*

Base class for all Services.

New in version 0.4: The *name* and *max_clients* keyword argument.

New in version 0.5: The *ignore_attributes* keyword argument.

New in version 1.0: If a method of the Service returns an object that is not natively JSON serializable, then the returned object can have a callable `to_json()` method and the value returned by `to_json()` will be used in the response to the *Client*.

Parameters

- **name** (`str`, optional) – The name of the Service as it will appear on the Network *Manager*. If not specified then the class name is used. You can also specify the *name* in the *start()* method.
- **max_clients** (`int`, optional) – The maximum number of *Client*s that can be linked with this Service. A value ≤ 0 or `None` means that there is no limit.
- **ignore_attributes** (`str` or `list` of `str`, optional) – The names of the attributes to not include in the *identity* of the Service. See *ignore_attributes()* for more details.

property max_clients

The maximum number of *Client*s that can be linked with this *Service*. A value ≤ 0 means an unlimited number of *Client*s can be linked.

Type

`int`

emit_notification(**args*, ***kwargs*)

Emit a notification to all *Client*s that are *Link*ed with this *Service*.

New in version 0.5.

Parameters

- **args** – The arguments to emit.
- **kwargs** – The keyword arguments to emit.

See also:

emit_notification_threadsafe(), *notification_handler()*

emit_notification_threadsafe(**args*, ***kwargs*)

A thread-safe implementation of *emit_notification()*.

When a *Service* handles a request, it does so in a separate thread than the event loop is running in. Therefore, if a method of the *Service* class wants to emit a notification while it is handling a request then it must emit the notification in a thread-safe manner.

New in version 1.0.

Parameters

- **args** – The arguments to emit.
- **kwargs** – The keyword arguments to emit.

See also:

emit_notification(), *notification_handler()*

ignore_attributes(**names*)

Ignore attributes from being added to the *identity* of the *Service*.

There are a few reasons why you may want to call this method:

- If you see warnings that an object is not JSON serializable or that the signature of an attribute cannot be found when starting the *Service* and you prefer not to see the warnings.
- If you do not want an attribute to be made publicly known that it exists. However, a *Client* can still access the ignored attributes.

Private attributes (i.e., attributes that start with an underscore) are automatically ignored and cannot be accessed from a *Client* on the network.

If you want to ignore any attributes then you must call *ignore_attributes()* before calling *start()*.

New in version 0.5.

Parameters
names – The names of the attributes to not include in the *identity* of the *Service*.

start(**, name=None, host='localhost', port=1875, timeout=10, username=None, password=None, password_manager=None, read_limit=None, disable_tls=False, cert_file=None, assert_hostname=True, auto_save=False*)

Start the *Service*.

See *connect()* for the description of each parameter.

msl.network.service.**filter_service_start_kwargs**(***kwargs*)

From the specified keyword arguments only return those that are valid for *start()*.

New in version 0.4.

Parameters
kwargs – All keyword arguments that are not part of the method signature for *start()* are silently ignored and are not included in the output.

Returns
dict – Valid keyword arguments that can be passed to *start()*.

msl.network.ssh module

Helper functions for connecting to a remote computer via SSH.

Follow these instructions to install/enable an SSH server on Windows. You can also create an SSH server using the paramiko package (which is included when MSL-Network is installed).

The two functions `start_manager()` and `parse_console_script_kwargs()` are meant to be used together to automatically start a Network `Manager`, and possibly `Service`s, on a remote computer.

See *Starting a Service from another computer* for an example on how to start a `Service` on a Raspberry Pi from another computer.

`msl.network.ssh.`**`parse_console_script_kwargs`**`()`

Parses the command line for keyword arguments sent from a remote computer.

New in version 0.4.

Returns
`dict` – The keyword arguments that were passed from `start_manager()`.

`msl.network.ssh.`**`start_manager`**`(`*host*, *console_script_path*, *, *ssh_username=None*, *ssh_password=None*, *timeout=10*, *as_sudo=False*, *missing_host_key_policy=None*, *paramiko_kwargs=None*, ***kwargs*`)`

Start a Network `Manager` on a remote computer.

New in version 0.4.

Parameters

- **host** (`str`) – The hostname (or IP address) of the remote computer. For example – `'192.168.1.100'`, `'raspberrypi'`, `'pi@raspberrypi'`
- **console_script_path** (`str`) – The file path to where the console script is located on the remote computer.
- **ssh_username** (`str`, optional) – The username to use to establish the SSH connection. If `None` and the *ssh_username* is not specified in *host* then you will be asked for the *ssh_username*.
- **ssh_password** (`str`, optional) – The password to use to establish the SSH connection. If `None` then you will be asked for the *ssh_password*.
- **timeout** (`int` or `float`, optional) – The maximum number of seconds to wait for the SSH connection.
- **as_sudo** (`bool`, optional) – Whether to run the console script as a superuser.
- **missing_host_key_policy** (`MissingHostKeyPolicy`, optional) – The policy to use when connecting to servers without a known host key. If `None` then uses `AutoAddPolicy`.
- **paramiko_kwargs** (`dict`, optional) – Additional keyword arguments that are passed to `ssh.connect`.
- **kwargs** – The keyword arguments in `run_forever()`, and if that console script also starts `Service`s on the remote computer as well, then the keyword arguments also found in `start()`. The *kwargs* should be parsed by `parse_console_script_kwargs()` on the remote computer.

`msl.network.ssh.`**`connect`**(*host*, ***, *username=None*, *password=None*, *timeout=10*, *missing_host_key_policy=None*, ***kwargs*)

SSH to a remote computer.

New in version 0.4.

Parameters

- **`host`** (`str`) – The hostname (or IP address) of the remote computer. For example – `'192.168.1.100'`, `'raspberrypi'`, `'pi@raspberrypi'`
- **`username`** (`str`, optional) – The username to use to establish the SSH connection. If `None` and the *username* is not specified in *host* then you will be asked for the *username*.
- **`password`** (`str`, optional) – The password to use to establish the SSH connection. If `None` then you will be asked for the *password*.
- **`timeout`** (`int` or `float`, optional) – The maximum number of seconds to wait for the SSH connection.
- **`missing_host_key_policy`** (`MissingHostKeyPolicy`, optional) – The policy to use when connecting to servers without a known host key. If `None` then uses `AutoAddPolicy`.
- **`kwargs`** – Additional keyword arguments that are passed to `SSHClient.connect`.

Returns

`SSHClient` – The SSH connection to the remote computer.

`msl.network.ssh.`**`exec_command`**(*ssh_client*, *command*, ***, *timeout=10*)

Execute the SSH command on the remote computer.

New in version 0.4.

Parameters

- **`ssh_client`** (`SSHClient`) – The SSH client that has already established a connection to the remote computer. See also *`connect()`*.
- **`command`** (`str`) – The command to execute on the remote computer.
- **`timeout`** (`int` or `float`, optional) – The maximum number of seconds to wait for the command to finish.

Raises

`RuntimeError` – If an error occurred. Either a timeout or stderr on the remote computer contains text from executing the *command*.

Returns

`list` of `str` – stdout from the remote computer.

msl.network.utils module

Common functions used by MSL-Network.

`msl.network.utils.`**`ensure_root_path`**`(`*path*`)`

Ensure that the root directory of the file path exists.

Parameters

`path` (`str`) – A file path. For example, if *path* is `/the/path/to/my/test/file.txt` then this function would ensure that the `/the/path/to/my/test` directories exist (creating the intermediate directories if necessary).

`msl.network.utils.`**`parse_terminal_input`**`(`*line*`)`

Parse text from a terminal connection.

See, *Connecting from a Terminal* for more details.

Parameters

`line` (`str`) – The input text from the terminal.

Returns

`dict` – The JSON object.

1.11 License

1.12 Developers

- Joseph Borbely <joseph.borbely@measurement.govt.nz>

1.13 Release Notes

1.13.1 Version 1.0.0 (2023-06-16)

- Added
  - a *Link* can create an exclusive or shared lock with a *Service*
  - add *service_max_clients* property to a *Link* and *LinkedClient*
  - the *loop_thread_id* property for a *Service* and a *Client*
  - the *emit_notification_threadsafe()* method for a *Service*
  - ability to specify the *host* to use when starting a *Manager*
  - support for Python 3.9, 3.10 and 3.11
  - *set_logging_level()* to be able to set the logging level at runtime
  - ability to add tasks to the event loop via the *add_tasks()* method
  - the *shutdown_handler()* method is called after the connection to the *Manager* is lost but before the event loop stops
  - ability to use all *Database* classes as a context manager (i.e., using a *with* statement)
  - the `--log-level` flag to the `start` command
  - the `delete` command-line argument to delete files that are created by MSL-Network
  - orjson as a JSON backend to *Package*
  - `JSON`, `UJSON`, `RAPIDJSON` and `SIMPLEJSON` are aliases for the JSON backends in *Package*
  - the `read_limit` keyword arguments to *connect()* and *start()*
  - the `auto_save` keyword argument to *connect()* and *get_ssl_context()*
  - the `digest_size` keyword argument to *generate_certificate()* and *get_fingerprint()*
  - the `name` and `extensions` keyword arguments to *generate_certificate()*,
  - `**kwargs` to *get_ssl_context()*
- Changed
  - the *result* object that is returned by a *Service* response can implement a callable `to_json()` method
  - the value of the `algorithm` keyword argument in *get_fingerprint()* can now also be of type `str`
  - renamed `uuid` to be `uid` in the JSON format

  - making an asynchronous request now returns a `concurrent.futures.Future` instance instead of an `asyncio.Future` instance
  - *Client* and *Service* are subclasses of *Device*
  - move the `utils.localhost_aliases` function to be defined as *LOCALHOST_ALIASES*
  - renamed the `Client.manager` method to *identities()*
  - renamed `certfile` to `cert_file` in *connect()*, *start()* and *get_ssl_context()*
  - can now change which JSON backend to use during runtime by calling the *use()* function
  - moved the `msl.network.constants.JSONPackage` class to *msl.network.json.Package*
  - renamed the command line arguments `--certfile` to `--cert-file`, `--keyfile` to `--key-file`, `--keyfile-password` to `--key-file-password`, and `--logfile` to `--log-file` for the `start` command
  - use `T` as the separator between the date and time in a *ConnectionsTable*
  - rename the keyword arguments `timestamp1` to `start` and `timestamp2` to `end` in *connections()*
  - the default filename for the certificate and key files now use `'localhost'` instead of the value of *HOSTNAME*
- Fixed
  - an `AttributeError` could be raised when generating the identity of a *Service*
  - can now handle multiple requests/replies contained within the same network packet
- Removed
  - Support for Python 3.5
  - the `MSLNetworkError` exception class (a `RuntimeError` is raised instead)
  - the `Service.set_debug` method
  - the `termination` and `encoding` class attributes of *Network*
  - the `send_pending_requests`, `raise_latest_error` and `wait` methods of a *Client*
  - the `--debug` flag from the `start` command
  - the `utils.new_selector_event_loop` function
  - the `constants.JSON` attribute
  - YAJL as a JSON backend option

1.13.2 Version 0.5.0 (2020-03-18)

- Added
  - support for Python 3.8
  - the *utils.new_selector_event_loop* function to create a new *asyncio.SelectorEventLoop*
  - the `--logfile` command line argument for the *start* command
  - a *Service* can emit notifications to all *Clients* that are linked with it
  - a *Service* now accepts an *ignore_attributes* keyword argument when it is instantiated and also has an *ignore_attributes* method
  - a *Link* can unlink from a *Service*
  - the *LinkedClient.client* property
  - `1.0.0.127.in-addr.arpa` as a localhost alias
- Changed
  - use `__package__` as the logger name
  - the *FileHandler* and *StreamHandler* that are added to the root logger now use a decimal point instead of a comma between the seconds and milliseconds values
  - renamed the *disconnect_service* method for a *Link* and a *Service* (which was added in version 0.4.0) to be *shutdown_service*
- Removed
  - the *Service._shutdown* method since it is no longer necessary to call this method from the *Service* subclass because shutting down happens automatically behind the scenes

1.13.3 Version 0.4.1 (2019-07-23)

- Added
  - `1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.ip6.arpa` as a localhost alias
- Changed
  - calling the *Client.manager(as_string=True)* method now prints the attributes analogous to how a *Client* would call the method of a *Service*
- Fixed
  - the *timeout* value for creating a *LinkedClient* is now the total time that it takes to connect to the Network *Manager* plus the time required to link with the *Service* (this fixes a race condition when starting a *Service* on a remote computer and then trying to link to the same *Service*)

1.13.4 Version 0.4.0 (2019-04-16)

- Added
  - the *ssh* module
  - a *LinkedClient* class
  - the *run_forever* (to start the *Manager*) and the *run_services* (to start the *Manager* and then start the *Services*) functions
  - the *filter_service_start_kwargs*, *filter_run_forever_kwargs* and *filter_client_connect_kwargs* functions
  - a *disconnect_service* method to *Link*
  - shorter argument name options for some CLI parameters
  - a *Service* now accepts *name* and *max_clients* as keyword arguments when it is instantiated
- Changed
  - the following CLI changes to argument names for the *certgen* command
    * `--key-path` became `--keyfile`
    * `--key-password` became `--keyfile-password`
  - the following CLI changes to argument names for the *keygen* command
    * `--path` became `--out`
  - the following CLI changes to argument names for the *start* command
    * `--cert` became `--certfile`
    * `--key` became `--keyfile`
    * `--key-password` became `--keyfile-password`
  - the *certificate* keyword argument for the *connect* and *get_ssl_context* functions and for the *Service.start* method was changed to *certfile*
  - the *as_yaml* keyword argument for the *Client.manager* method was changed to *as_string*
  - a *Client* can no longer request a private attribute – i.e., an attribute that starts with a _ (an underscore) – from a *Service*
  - the default *timeout* value for connecting to the *Manager* is now 10 seconds
- Fixed
  - perform error handling if the *Manager* attempts to start on a port that is already in use
  - issue #7 - a *Service* can now specify the maximum number of *Clients* that can be linked with it
  - issue #6 - the *password_manager* keyword argument is now used properly when starting a *Service*
- Removed
  - the *name* class attribute for a *Service*
  - the *send_request* method for a *Client* (must link with a *Service*)

1.13.5 Version 0.3.0 (2019-01-06)

- Added
  - every request from a *Client* can now specify a timeout value
  - the docs now include an example for how to send requests to the `Echo` *Service*
- Changed
  - the default *timeout* value for connecting to the *Manager* is now 10 seconds
  - the *__repr__* method for a *Client* no longer includes the id as a hex number
- Fixed
  - issue #5
  - issue #4
  - issue #3
  - issue #2
  - issue #1
- Removed
  - the *__repr__* method for a *Service*

1.13.6 Version 0.2.0 (2018-08-24)

- Added
  - a `wakeup()` Task in debug mode on Windows (see: https://bugs.python.org/issue23057)
  - the `version_info` named tuple now includes a *releaselevel*
  - example for creating a *Client* and a *Service* in LabVIEW
  - the ability to establish a connection to the Network *Manager* without using TLS
  - a `timeout` kwarg to *Service.start()*
  - an `Echo` *Service* to the examples
- Changed
  - rename 'async' kwarg to be 'asynchronous' (for Python 3.7 support)
  - the termination bytes were changed from `\n` to `\r\n`

1.13.7 Version 0.1.0 (2017-12-14)

- Initial release

CHAPTER

TWO

INDEX

- modindex

PYTHON MODULE INDEX

INDEX