Gridworks Proactor

Jessica Millar

Apr 23, 2024

CONTENTS

[][license]

This packages provides “live actor” and “application monitored communication” infrastructure for the GridWorks SpaceHeat SCADA project. This separation allows the scada code to be more focussed on on application specific details and provides the potential to re-use the “live actor” and “application monitored” infrastructure.

CHAPTER

ONE

FEATURES

- *Proactor*, a single threaded event loop running on asyncio, for exchanging messages between the main application object, "live actor" subobjects and MQTT clients.
- A [communication state] ("active" or not) for each external communications link is available to the proactor and sub-objects. "Active" communications is defined as ALL of the following:
  - The underlying communications mechanism (MQTT) is connected.
  - All input channels of underlying mechanism (MQTT topics) are established.
  - A application messages requiring acknowledgement have been ACKed in timely fashion (by default 5 seconds).
  - A message has been received "recently" (by default within 1 minute).
- Reliable delievery of "Events" generated locally. Generated Events are stored locally until they are acknowledged and unacknowledged Events are retransmitted when the "Active" communication state is restored.
- *gwproactor_test*, a test package for development and test environments of projects that implement a class derived from *Proactor*, allowing the derived class to be tested with the base-class tests.

CHAPTER

TWO

REQUIREMENTS

2.1 Mosquitto

Testing requires an MQTT broker. The Mosquitto broker can be installed with:

```
brew install mosquitto
brew services restart mosquitto
```

2.2 TLS

Testing uses TLS by default. The tests require the path to the CA certificate and private key used to sign the certificate of the MQTT broker. To set up TLS:

Install gridworks-cert (gwcert):

```
pipx install gridworks-cert
```

Create a local Certificate Authority:

```
gwcert ca create
```

Create certificate and key for the Mosquitto MQTT broker:

```
gwcert key add --dns localhost mosquitto
```

- **NOTE**: This command will generate a broker certificate that *only* allow connections to `localhost`. See *External connections* below to create a broker certificate which can accept connections from external devices.

Find the path to `mosquitto.conf` in the output of:

```
brew services info mosquitto -v
```

Modify `mosquitto.conf` with the TLS configuration in example-test-mosquitto.conf, fixing up the paths with real absolute paths to certificate, key and CA certificate files. These paths can be found with:

```
gwcert ca info
```

Restart the Mosquitto server:

```
brew services restart mosquitto
```

Test Mosquitto 'clear' port:

```
# in one window
mosquitto_sub -h localhost -p 1883 -t foo
# in another window
mosquitto_pub -h localhost -p 1883 -t foo -m '{"bla":1}'
```

Test Mosquitto TLS port:

```
gwcert key add pubsub
# in one window
mosquitto_sub -h localhost -p 8883 -t foo \
     --cafile $HOME/.local/share/gridworks/ca/ca.crt \
     --cert $HOME/.local/share/gridworks/ca/certs/pubsub/pubsub.crt \
     --key $HOME/.local/share/gridworks/ca/certs/pubsub/private/pubsub.pem
# in another window
mosquitto_pub -h localhost -p 8883 -t foo \
     --cafile $HOME/.local/share/gridworks/ca/ca.crt \
     --cert $HOME/.local/share/gridworks/ca/certs/pubsub/pubsub.crt \
     --key $HOME/.local/share/gridworks/ca/certs/pubsub/private/pubsub.pem \
     -m '{"bar":1}'
```

2.2.1 Troubleshooting Mosquitto

Mosquitto logging can be enabled in the `mosquitto.conf` file with the lines:

```
log_dest stderr
log_type all
```

To see the console output, stop the Mosquitto service and start it explicitly on the command line:

```
brew services stop mosquitto
mosquitto -c /opt/homebrew/etc/mosquitto/mosquitto.conf
```

2.2.2 External connections

The broker certificate must be created with the *hostname* the client will use to connect to it. For example, to create a broker certificate reachable at `localhost`, `MyMac.local`, `192.168.1.10` and `foo.bar.baz` use the command:

```
gwcert key add \
  --dns localhost \
  --dns MyMac.local \
  --dns 192.168.1.10 \
  --dns foo.bar.baz \
  mosquitto
```

2.2.3 Pre-existing key files

If CA or Mosquito certificate can key files *already* exist, their paths can be specified in `mosquitto.conf` as above and for the tests with there GWPROACTOR_TEST_CA_CERT_PATH and GWPROACTOR_TEST_CA_KEY_PATH environment variables.

2.2.4 Disabling TLS

To disable testing of TLS, modify the the file `tests/.env-gwproactor-test` with:

```
GWCHILD_PARENT_MQTT__TLS__USE_TLS=false
GWPARENT_CHILD_MQTT__TLS__USE_TLS=false
```

CHAPTER

THREE

INSTALLATION

You can install *Gridworks Proactor* via pip from PyPI:

```
$ pip install gridworks-proactor
```

CHAPTER

FOUR

CONTRIBUTING

Contributions are very welcome. In order to develop, do this:

```
$ poetry install --all-extras
```

To learn more, see the [Contributor Guide].

CHAPTER

FIVE

LICENSE

Distributed under the terms of the [MIT license][license], *Gridworks Proactor* is free and open source software.

CHAPTER

SIX

ISSUES

If you encounter any problems, please file an issue along with a detailed description.

CHAPTER

SEVEN

CREDITS

This project was generated from @cjolowicz's Hypermodern Python Cookiecutter template.

7.1 Communication state

The `Proactor` maintains communication state ("active" or not active) for each external point-to-point communications link. The "active" state is intended to indicate that not only is the underlying communications channel (e.g. MQTT) healthy, but also that a valid application-level message has been recently received from the peer at the other end of the communications link. This state information is intended to allow the application derived from the Proactor to determine if it must make local decisions while the peer is disconnected, non-responsive, slow or otherwise impaired. Additionally, visibility into the history of communication is provided to (human) monitors of the system through Events generated at each transition of the the comm state.

7.1.1 "active" communication state definition

A communication link is "active" if *all* of these are true:

1. The underlying communications mechanism (MQTT) is connected.
2. All input channels of underlying mechanism (MQTT topics) are established.
3. All application messages requiring acknowledgement have been ACKed in timely fashion (by default 5 seconds).
4. A valid message has been received "recently" (by default within 1 minute) from the peer application.

Note after the underying communication mechanism reports a connection, before communication can be considered "active", requirements 2 and 4 above must be met. That is, all input channels must be established and at least one valid application message must be received from the peer. Requirement 2 is present because otherwise we could send a message but not hear the response to it from the peer. Requirement 4 is present because we could have good underlying communication (e.g. a connection to an MQTT broker), without the peer application actually running. Requirement 3 is not applied until after the "active" state has been reached.

This diagram approximates how the "active" state is achieved, maintained and lost:

Much of the complexity in this diagram results from asynchronously accumulating input channel establishments and a message from the peer upon restore of the underlying connection. After restoring communication to the underlying communication mechanism (e.g. an MQTT broker), we must get acknowledgements of all our subscriptions and a message from the peer before the link is considered "active". There could be more than one subscription acknowledgement message, and these and the message from the peer could arrive in any order. This complexity could be reduced by serializing the accumulation of these results, at the cost of longer time to re-activate after restore of the underlying communication mechanism.

7.1.2 LinkManager

The *gwproactor.links* package implements most of the Proactor's communication infrastructure. *LinkManager* is the interface to this package used by *Proactor*. The interaction between them can be seen by searching the code for `_links`. This search should produce approximately the following entry points in the message processing loop:

1. Start on user request.
2. Stop on user request (omitted from the diagram for clarity).
3. Handle connect of underlying comm mechanism (e.g. broker connect of MQTT).
4. Handle disconnect of underlying comm mechanism.
5. Handle intermediate connection establishment events from underlying comm mechanism (e.g. subscription ack of MQTT).
6. Send acks for incoming messages that require ack.
7. Receive acks for outgoing messages that require acks.
8. Handle timeouts for ack receipt.
9. Update "heard from recently" on message receipt.
10. Handle timeouts for "heard from recently".

LinkManager helpers

The *LinkManager* uses these helpers:

- , to manage Paho MQTT clients.
- a dict of MQTTCodec, to contain a message coder/decoder for each MQTT client.
- *LinkStates*, to manage the communications state machine for each link.
- *MessageTimes*, to track the times of last send and receive for each link.
- *TimerManagerInterface*, to start and cancel timers for acknowledgement timeout.
- *AckManager*, to start, track, handle and cancel timers for pending acknowledgements.
- *PersisterInterface*, to persist unacknowledged Events on loss of "active" communication and re-upload them when "active" state is restored.
- *ProactorLogger*, to log communications state tranisitions.
- *ProactorStats*, to update various statistics about communications.

7.2 Reference

- *gwroactor*, the package interface.
  - *gwproactor.links*, an internal package used to manage communication state.
- *gwroactor_test*, a development package, used to test classes inherting from `Proactor`

7.2.1 gwproactor

This packages provides infrastructure for running a proactor on top of asyncio with support multiple MQTT clients and and sub-objects which support their own threads for synchronous operations.

This packages is not GridWorks-aware (except that it links actors with multiple mqtt clients). This separation between communication / action infrastructure and GridWorks semantics is intended to allow the latter to be more focussed.

This package is not polished and the separation is up for debate.

Particular questions:

- Is the programming model still clean after more concrete actors are implemented and more infrastructure are added.
- Does the separation add value or just complicate analysis.
- MQTTClients should be made async.
- Semantics of building message type namespaces should be spelled out / further worked out.
- Test support should be implemented / cleaner.

class gwproactor.**Actor**(*name*, *services*)

Parameters

- **name** (*str*) –
- **services** (ServicesInterface) –

property alias

init()

Called after constructor so derived functions can be used in setup.

Return type
None

property node

class gwproactor.**ActorInterface**

Pure interface for a proactor sub-object (an Actor) which can communicate and has a GridWorks ShNode.

abstract property alias: str

abstract init()

Called after constructor so derived functions can be used in setup.

Return type
None

classmethod load(*name*, *actor_class_name*, *services*, *module_name*)

Parameters

- **name** (*str*) –
- **actor_class_name** (*str*) –
- **services** (ServicesInterface) –
- **module_name** (*str*) –

Return type
ActorInterface

abstract property node: ShNode

class gwproactor.**AsyncQueueWriter**

Allow synchronous code to write to an asyncio Queue.

It is assumed the asynchronous reader has access to the asyncio Queue "await get()" from directly from it.

put(*item*)

Write to asyncio queue in a threadsafe way.

Parameters
item (*Any*) –

Return type
None

set_async_loop(*loop*, *async_queue*)

Parameters

- **loop** (*AbstractEventLoop*) –
- **async_queue** (*Queue*) –

Return type
None

class gwproactor.**Communicator**(*name*, *services*)

A partial implementation of CommunicatorInterface which supplies the trivial implementations

Parameters

- **name** (*str*) –
- **services** (ServicesInterface) –

property monitored_names: Sequence[*MonitoredName*]

property name: str

property services: *ServicesInterface*

class gwproactor.**CommunicatorInterface**

Pure interface necessary for interaction between a sub-object and the system services proactor

abstract property monitored_names: Sequence[*MonitoredName*]

abstract property name: str

abstract process_message(*message*)

Parameters
message (*Message*) –

Return type
Ok[bool] | *Err*[BaseException]

abstract property services: *ServicesInterface*

class gwproactor.**ExternalWatchdogCommandBuilder**

Create arguments which will be passed to subprocess.run() to pat the external watchdog.

If the returned list is empty, pat process will be run.

By default an empty list is returned if the environment variable named by sevice_variable_name() is not set to 1 or true.

classmethod default_pat_args(*pid=None*)

Parameters
pid (*int | None*) –

Return type
list[str]

classmethod pat_args(*service_name*, *args=None*, *pid=None*)

Return arguments to be passed to subprocess.run() to pat the external watchdog.

Parameters

- **service_name** (*str*) –
- **args** (*list[str] | None*) –
- **pid** (*int | None*) –

Return type
list[str]

classmethod running_as_service(*service_name*)

Parameters
service_name (*str*) –

Return type
bool

classmethod service_variable_name(*service_name*)

Parameters
service_name (*str*) –

Return type
str

class gwproactor.**MQTTClientWrapper**(*name*, *client_config*, *receive_queue*)

Parameters

- **name** (*str*) –
- **client_config** (*MQTTClient*) –
- **receive_queue** (AsyncQueueWriter) –

connected()

Return type
bool

disable_logger()

enable_logger(*logger=None*)

Parameters
logger (*Logger | LoggerAdapter | None*) –

handle_suback(*suback*)

Parameters
suback (*MQTTSubackPayload*) –

Return type
int

num_pending_subscriptions()

Return type
int

num_subscriptions()

Return type
int

on_connect(*_*, *userdata*, *flags*, *rc*)

on_connect_fail(*_*, *userdata*)

on_disconnect(*_*, *userdata*, *rc*)

on_message(*_*, *userdata*, *message*)

on_subscribe(*_*, *userdata*, *mid*, *granted_qos*)

publish(*topic*, *payload*, *qos*)

Parameters

- **topic** (*str*) –
- **payload** (*bytes*) –
- **qos** (*int*) –

Return type
MQTTMessageInfo

start()

stop()

subscribe(*topic*, *qos*)

Parameters

- **topic** (*str*) –
- **qos** (*int*) –

Return type
Tuple[int, int | None]

subscribe_all()

Return type
Tuple[int, int | None]

subscribed()

Return type
bool

subscription_items()

Return type
list[*Tuple*[str, int]]

unsubscribe(*topic*)

Parameters
topic (*str*) –

Return type
Tuple[int, int | None]

class gwproactor.**MQTTClients**

add_client(*name*, *client_config*, *upstream=False*, *primary_peer=False*)

Parameters

- **name** (*str*) –
- **client_config** (*MQTTClient*) –
- **upstream** (*bool*) –
- **primary_peer** (*bool*) –

client_wrapper(*client*)

Parameters
client (*str*) –

Return type
MQTTClientWrapper

clients: Dict[str, *MQTTClientWrapper*]

connected(*client*)

Parameters
client (*str*) –

Return type
bool

disable_loggers()

enable_loggers(*logger=None*)

Parameters
logger (*Logger | LoggerAdapter | None*) –

handle_suback(*suback*)

Parameters
suback (*MQTTSubackPayload*) –

Return type
int

num_pending_subscriptions(*client*)

Parameters
client (*str*) –

Return type
int

num_subscriptions(*client*)

Parameters
client (*str*) –

Return type
int

primary_peer()

Return type
MQTTClientWrapper

primary_peer_client: str = ''

publish(*client*, *topic*, *payload*, *qos*)

Parameters

- **client** (*str*) –
- **topic** (*str*) –
- **payload** (*bytes*) –
- **qos** (*int*) –

Return type
MQTTMessageInfo

start(*loop*, *async_queue*)

Parameters

- **loop** (*AbstractEventLoop*) –
- **async_queue** (*Queue*) –

stop()

subscribe(*client*, *topic*, *qos*)

Parameters

- **client** (*str*) –
- **topic** (*str*) –
- **qos** (*int*) –

Return type
Tuple[int, int | None]

subscribe_all(*client*)

Parameters
client (*str*) –

Return type
Tuple[int, int | None]

subscribed(*client*)

Parameters
client (*str*) –

Return type
bool

unsubscribe(*client*, *topic*)

Parameters

- **client** (*str*) –
- **topic** (*str*) –

Return type
Tuple[int, int | None]

upstream()

Return type
MQTTClientWrapper

upstream_client: str = ''

class gwproactor.**MonitoredName**(*name: str*, *timeout_seconds: float*)

Parameters

- **name** (*str*) –
- **timeout_seconds** (*float*) –

name: str

timeout_seconds: float

class gwproactor.**Proactor**(*name*, *settings*, *hardware_layout=None*)

Parameters

- **name** (*str*) –
- **settings** (ProactorSettings) –
- **hardware_layout** (*HardwareLayout | None*) –

add_communicator(*communicator*)

Parameters
communicator (CommunicatorInterface) –

add_web_route(*server_name*, *method*, *path*, *handler*, ***kwargs*)

Adds configuration for web server route which will be available after start() is called.

May be called even if associated web server is not configured, in which case this route will simply be ignored.

Not thread safe.

Parameters

- **server_name** (*str*) –
- **method** (*str*) –
- **path** (*str*) –
- **handler** (*Callable[[Request], Awaitable[StreamResponse]]*) –
- **kwargs** (*Any*) –

add_web_server_config(*name*, *host*, *port*, ***kwargs*)

Adds configuration for web server which will be started when start() is called.

Not thread safe.

Parameters

- **name** (*str*) –
- **host** (*str*) –
- **port** (*int*) –
- **kwargs** (*Any*) –

Return type
None

property async_receive_queue: Queue | None

property event_loop: AbstractEventLoop | None

generate_event(*event*)

Parameters
event (*EventT*) –

Return type
Ok[bool] | *Err*[BaseException]

get_communicator(*name*)

Parameters
name (*str*) –

Return type
CommunicatorInterface | None

get_communicator_as_type(*name*, *type_*)

Parameters

- **name** (*str*) –
- **type_** (*Type[T]*) –

Return type
T | None

get_external_watchdog_builder_class()

Return type
type[ExternalWatchdogCommandBuilder]

property hardware_layout: HardwareLayout

property io_loop_manager: IOLoopInterface

async join()

property logger: *ProactorLogger*

classmethod make_event_persister(*settings*)

Parameters
settings (ProactorSettings) –

Return type
PersisterInterface

classmethod make_stats()

Return type
ProactorStats

property monitored_names: Sequence[*MonitoredName*]

property name: str

property primary_peer_client: str

async process_message(*message*)

Parameters
message (*Message*) –

async process_messages()

property publication_name: str

async run_forever()

send(*message*)

Parameters
message (*Message*) –

send_threadsafe(*message*)

Parameters
message (*Message*) –

Return type
None

property services: *ServicesInterface*

property settings: *ProactorSettings*

start()

start_tasks()

property stats: ProactorStats

stop()

property upstream_client: str

class gwproactor.ProactorLogger(*base*, *message_summary*, *lifecycle*, *comm_event*, *extra=None*)

Parameters

- **base** (*str*) –
- **message_summary** (*str*) –
- **lifecycle** (*str*) –
- **comm_event** (*str*) –
- **extra** (*dict | None*) –

MESSAGE_DELIMITER_WIDTH = 88

MESSAGE_ENTRY_DELIMITER =
'++'

MESSAGE_EXIT_DELIMITER =
'--'

comm_event(*msg*, **args*, ***kwargs*)

Parameters
msg (*str*) –

Return type
None

property comm_event_enabled: bool

comm_event_logger: Logger

property general_enabled: bool

lifecycle(*msg*, **args*, ***kwargs*)

Parameters
msg (*str*) –

Return type
None

property lifecycle_enabled: bool

lifecycle_logger: Logger

message_enter(*msg*, **args*, ***kwargs*)

Parameters
msg (*str*) –

Return type
None

message_exit(*msg*, **args*, ***kwargs*)

Parameters
msg (*str*) –

Return type
None

message_summary(*direction*, *actor_alias*, *topic*, *payload_object=None*, *broker_flag=' '*, *timestamp=None*, *message_id=''*)

Parameters

- **direction** (*str*) –
- **actor_alias** (*str*) –
- **topic** (*str*) –
- **payload_object** (*Any*) –
- **broker_flag** (*str*) –
- **timestamp** (*datetime | None*) –
- **message_id** (*str*) –

Return type
None

property message_summary_enabled: bool

message_summary_logger: Logger

path(*msg*, **args*, ***kwargs*)

Parameters
msg (*str*) –

Return type
None

property path_enabled: bool

class gwproactor.**ProactorSettings**(*_env_file='<object object>'*, *_env_file_encoding=None*, *_env_nested_delimiter=None*, *_secrets_dir=None*, ***, *paths=None*, *logging=LoggingSettings(base_log_name='gridworks', base_log_level=30, levels=LoggerLevels(message_summary=30, lifecycle=20, comm_event=20), formatter=FormatterSettings(fmt='%(asctime)s %(message)s', datefmt='', default_msec_format='%s.%03d'), file_handler=RotatingFileHandlerSettings(filename='proactor.log', bytes_per_log_file=2097152, num_log_files=10, level=0))*, *mqtt_link_poll_seconds=60.0*, *ack_timeout_seconds=5.0*, *num_initial_event_reuploads=5*)

Parameters

- **_env_file** (*str | PathLike | List[str | PathLike] | Tuple[str | PathLike, ...] | None*) –

- **_env_file_encoding** (*str | None*) –
- **_env_nested_delimiter** (*str | None*) –
- **_secrets_dir** (*str | PathLike | None*) –
- **paths** (*Paths*) –
- **logging** (*LoggingSettings*) –
- **mqtt_link_poll_seconds** (*float*) –
- **ack_timeout_seconds** (*float*) –
- **num_initial_event_reuploads** (*int*) –

class Config

env_nested_delimiter = '__'

env_prefix = 'PROACTOR_'

ack_timeout_seconds: float

classmethod get_paths(*v*)

Parameters
v (*Paths*) –

Return type
Paths

logging: LoggingSettings

mqtt_link_poll_seconds: float

num_initial_event_reuploads: int

paths: Paths

classmethod post_root_validator(*values*)

Update unset paths of any member MQTTClient's TLS paths based on ProactorSettings 'paths' member.

Parameters
values (*dict*) –

Return type
dict

classmethod update_paths_name(*values*, *name*)

Update paths member with a new 'name' attribute, e.g., a name known by a derived class.

This is meant to be called in a 'pre=True' root validator of a derived class.

Parameters

- **values** (*dict*) –
- **name** (*str*) –

Return type
dict

exception gwproactor.Problems(*msg=''*, *warnings=None*, *errors=None*, *max_problems=10*)

Parameters

- **msg** (*str*) –
- **warnings** (*list[BaseException]*) –
- **errors** (*list[BaseException]*) –
- **max_problems** (*int | None*) –

MAX_PROBLEMS = 10

add_error(*error*)

Parameters
error (*BaseException*) –

Return type
Problems

add_problems(*other*)

Parameters
other (Problems) –

Return type
Problems

add_warning(*warning*)

Parameters
warning (*BaseException*) –

Return type
Problems

error_traceback_str()

Return type
str

errors: list[BaseException]

max_problems: int | None = 10

problem_event(*summary*, *src=''*)

Parameters

- **summary** (*str*) –
- **src** (*str*) –

Return type
ProblemEvent

warnings: list[BaseException]

class gwproactor.QOS(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

AtLeastOnce = 1

AtMostOnce = 0

ExactlyOnce = 2

class gwproactor.**Runnable**

Pure interface to an object which is expected to support starting, stopping and joining.

abstract async join()

Return type
None

abstract start()

Return type
None

abstract stop()

Return type
None

async stop_and_join()

Return type
None

class gwproactor.**ServicesInterface**

Interface to system services (the proactor)

abstract add_web_route(*server_name*, *method*, *path*, *handler*, ***kwargs*)

Adds configuration for web server route which will be available after start() is called.

May be called even if associated web server is not configured, in which case this route will simply be ignored.

Not thread safe.

Parameters

- **server_name** (*str*) –
- **method** (*str*) –
- **path** (*str*) –
- **handler** (*Callable[[Request], Awaitable[StreamResponse]]*) –
- **kwargs** (*Any*) –

abstract add_web_server_config(*name*, *host*, *port*, ***kwargs*)

Adds configuration for web server which will be started when start() is called.

Not thread safe.

Parameters

- **name** (*str*) –
- **host** (*str*) –
- **port** (*int*) –
- **kwargs** (*Any*) –

Return type
None

abstract property async_receive_queue: Queue | None

abstract property event_loop: AbstractEventLoop | None

abstract generate_event(*event*)

Parameters
event (*EventT*) –

Return type
None

abstract get_communicator(*name*)

Parameters
name (*str*) –

Return type
CommunicatorInterface | None

abstract get_communicator_as_type(*name*, *type_*)

Parameters

- **name** (*str*) –
- **type_** (*Type[T]*) –

Return type
T | None

abstract get_external_watchdog_builder_class()

Return type
type[ExternalWatchdogCommandBuilder]

abstract property hardware_layout: HardwareLayout

abstract property io_loop_manager: IOLoopInterface

abstract property logger: *ProactorLogger*

abstract property publication_name: str

abstract send(*message*)

Parameters
message (*Message*) –

Return type
None

abstract send_threadsafe(*message*)

Parameters
message (*Message*) –

Return type
None

abstract property settings: *ProactorSettings*

abstract property stats: ProactorStats

class gwproactor.**Subscription**(*Topic*, *Qos*)

Parameters

- **Topic** (*str*) –
- **Qos** (QOS) –

Qos: *QOS*

Alias for field number 1

Topic: str

Alias for field number 0

class gwproactor.**SyncAsyncInteractionThread**(*channel=None*, *name=None*, *iterate_sleep_seconds=None*, *responsive_sleep_step_seconds=0.1*, *pat_timeout=20*, *daemon=True*)

A thread wrapper providing an async-sync communication channel and simple "iterate, sleep, read message" semantics.

Parameters

- **channel** (SyncAsyncQueueWriter | *None*) –
- **name** (*str* | *None*) –
- **iterate_sleep_seconds** (*float* | *None*) –
- **responsive_sleep_step_seconds** (*float*) –
- **pat_timeout** (*float* | *None*) –
- **daemon** (*bool*) –

JOIN_CHECK_THREAD_SECONDS = 1.0

PAT_TIMEOUT = 20

SLEEP_STEP_SECONDS = 0.1

async async_join(*timeout=None*)

Parameters
timeout (*float*) –

Return type
None

pat_timeout: float | None

pat_watchdog()

put_to_sync_queue(*message*, *block=True*, *timeout=None*)

Parameters

- **message** (*Any*) –
- **block** (*bool*) –
- **timeout** (*float* | *None*) –

Return type
None

`request_stop`()

Return type
None

`run`()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

`running: bool | None`

`set_async_loop`(*loop*, *async_queue*)

Parameters

- **loop** (*AbstractEventLoop*) –
- **async_queue** (*Queue*) –

Return type
None

`set_async_loop_and_start`(*loop*, *async_queue*)

Parameters

- **loop** (*AbstractEventLoop*) –
- **async_queue** (*Queue*) –

Return type
None

`time_to_pat`()

Return type
bool

`class gwproactor.SyncAsyncQueueWriter`(*sync_queue=None*)

Provide a full duplex communication "channel" between synchronous and asynchronous code.

It is assumed the asynchronous reader has access to the asyncio Queue "await get()" from directly from it.

Parameters
sync_queue (*Queue | None*) –

`get_from_sync_queue`(*block=True*, *timeout=None*)

Read from synchronous queue in a threadsafe way.

Parameters

- **block** (*bool*) –
- **timeout** (*float | None*) –

Return type
Any

put_to_async_queue(*item*)

Write to asynchronous queue in a threadsafe way.

Parameters
item (*Any*) –

put_to_sync_queue(*item*, *block=True*, *timeout=None*)

Write to synchronous queue in a threadsafe way.

Parameters

- **item** (*Any*) –
- **block** (*bool*) –
- **timeout** (*float | None*) –

set_async_loop(*loop*, *async_queue*)

Parameters

- **loop** (*AbstractEventLoop*) –
- **async_queue** (*Queue*) –

Return type
None

sync_queue: Queue | None

class gwproactor.**SyncThreadActor**(*name*, *services*, *sync_thread*)

Parameters

- **name** (*str*) –
- **services** (ServicesInterface) –
- **sync_thread** (SyncAsyncInteractionThread) –

async join()

property monitored_names: Sequence[*MonitoredName*]

process_message(*message*)

Parameters
message (*Message*) –

Return type
Ok[bool] | *Err*[BaseException]

send_driver_message(*message*)

Parameters
message (*Any*) –

Return type
None

start()

stop()

gwproactor.**format_exceptions**(*exceptions*)

Parameters
exceptions (*list[BaseException]*) –

Return type
str

gwproactor.**responsive_sleep**(*obj*, *seconds*, *step_duration=0.1*, *running_field_name='_main_loop_running'*, *running_field=True*)

Sleep in way that is more responsive to thread termination: sleep in step_duration increments up to specificed seconds, at after each step checking obj._main_loop_running. If the designated running_field_name actually indicates that a stop has been requested (e.g. what you would expect from a field named '_stop_requested'), set running_field parameter to False.

Parameters

- **seconds** (*float*) –
- **step_duration** (*float*) –
- **running_field_name** (*str*) –
- **running_field** (*bool*) –

Return type
bool

gwproactor.**setup_logging**(*args*, *settings*, *errors=None*, *add_screen_handler=True*, *root_gets_handlers=True*)

Get python logging config based on parsed command line args, defaults, environment variables and logging config file.

The order of precedence is:

1. Command line arguments
2. Environment
3. Defaults

Parameters

- **args** (*Namespace*) –
- **settings** (ProactorSettings) –
- **errors** (*list[BaseException] | None*) –
- **add_screen_handler** (*bool*) –
- **root_gets_handlers** (*bool*) –

Return type
None

7.2.2 gwproactor.links

Internal package used to manage communications state. *LinkManager* is the interface into this package used by *Proactor*

class gwproactor.links.**AckManager**(*timer_mgr*, *callback*, *delay=5.0*)

Parameters

- **timer_mgr** (TimerManagerInterface) –
- **callback** (*Callable[[AckWaitInfo], None]*) –
- **delay** (*float | None*) –

_acks: dict[str, dict[str, *AckWaitInfo*]]

_callback: Callable[[*AckWaitInfo*], None]

_default_delay_seconds: float

_pop_wait_info(*link_name*, *message_id*)

Parameters

- **link_name** (*str*) –
- **message_id** (*str*) –

Return type
AckWaitInfo | None

_timeout(*link_name*, *message_id*)

Parameters

- **link_name** (*str*) –
- **message_id** (*str*) –

Return type
None

_timer_mgr: *TimerManagerInterface*

add_link(*link_name*)

Parameters
link_name (*str*) –

Return type
None

cancel_ack_timer(*link_name*, *message_id*)

Parameters

- **link_name** (*str*) –
- **message_id** (*str*) –

Return type
AckWaitInfo

cancel_ack_timers(*link_name*)

Parameters
link_name (*str*) –

Return type
list[AckWaitInfo]

num_acks(*link_name*)

Parameters
link_name (*str*) –

Return type
int

start_ack_timer(*link_name*, *message_id*, *context=None*, *delay_seconds=None*)

Parameters

- **link_name** (*str*) –
- **message_id** (*str*) –
- **context** (*Any | None*) –
- **delay_seconds** (*float | None*) –

Return type
AckWaitInfo

class gwproactor.links.**AckWaitInfo**(*link_name: str*, *message_id: str*, *timer_handle: Any*, *context: Any = None*)

Parameters

- **link_name** (*str*) –
- **message_id** (*str*) –
- **timer_handle** (*Any*) –
- **context** (*Any*) –

context: Any = None

link_name: str

message_id: str

timer_handle: Any

class gwproactor.links.**AsyncioTimerManager**

_abc_impl = <_abc._abc_data object>

cancel_timer(*timer_handle*)

Cancel callback associated with _timer_handle_.

Note that callback might still run after this call returns.

Parameters
timer_handle (*Any*) – The value returned by start_timer()

Return type
None

start_timer(*delay_seconds*, *callback*)

Start a timer. Implementation is expected to call _callback_ after approximately _delay_sceonds_.

The execution context (e.g. the thread) of the callback must be specified by the implemntation.

The callback must have sufficient context available to it do its work as well as to detect if it is no longer relevant. Note a callback might run after cancelation if the callack was already "in-flight" at time of cancellation and it is up to the callback to tolerate this situation.

Parameters

- **delay_seconds** (*float*) – The approximate delay before the callback is called.
- **callback** (*Callable[[], None]*) – The function called after delay_seconds.

Returns
A timer handle which can be passed to _cancel_timer()_ to cancel the callback.

Return type
TimerHandle

exception gwproactor.links.**CommLinkAlreadyExists**(*name=''*, *current_state=StateName.none*, *transition=TransitionName.none*, ***, *msg=''*)

Parameters

- **name** (*str*) –
- **current_state** (StateName) –
- **transition** (TransitionName) –
- **msg** (*str*) –

exception gwproactor.links.**CommLinkMissing**(*name*, ***, *msg=''*)

Parameters
name (*str*) –

exception gwproactor.links.**InvalidCommStateInput**(*name=''*, *current_state=StateName.none*, *transition=TransitionName.none*, ***, *msg=''*)

Parameters

- **name** (*str*) –
- **current_state** (StateName) –
- **transition** (TransitionName) –
- **msg** (*str*) –

current_state: *StateName* = 'none'

name: str = ''

transition: *TransitionName* = 'none'

class gwproactor.links.**LinkManager**(*publication_name*, *settings*, *logger*, *stats*, *event_persister*, *timer_manager*, *ack_timeout_callback*)

Parameters

- **publication_name** (*str*) –
- **settings** (ProactorSettings) –
- **logger** (ProactorLogger) –
- **stats** (*ProactorStats*) –
- **event_persister** (*PersisterInterface*) –
- **timer_manager** (TimerManagerInterface) –
- **ack_timeout_callback** (*Callable[[AckWaitInfo], None]*) –

PERSISTER_ENCODING = 'utf-8'

_acks: *AckManager*

_event_persister: PersisterInterface

_logger: *ProactorLogger*

_message_times: *MessageTimes*

_mqtt_clients: *MQTTClients*

_mqtt_codecs: dict[str, MQTTCodec]

_recv_activated(*transition*)

Parameters
transition (Transition) –

_reupload_events(*event_ids*)

Parameters
event_ids (*list[str]*) –

Return type
Ok[bool] | *Err*[BaseException]

_reuploads: *Reuploads*

_settings: *ProactorSettings*

_start_reupload()

Return type
None

_states: *LinkStates*

_stats: ProactorStats

add_mqtt_link(*name*, *mqtt_config*, *codec=None*, *upstream=False*, *primary_peer=False*)

Parameters

- **name** (*str*) –
- **mqtt_config** (*MQTTClient*) –
- **codec** (*MQTTCodec | None*) –
- **upstream** (*bool*) –

- **primary_peer** (*bool*) –

decode(*link_name*, *topic*, *payload*)

Parameters

- **link_name** (*str*) –
- **topic** (*str*) –
- **payload** (*bytes*) –

Return type
Message[Any]

decoder(*link_name*)

Parameters
link_name (*str*) –

Return type
MQTTCodec | None

disable_mqtt_loggers()

enable_mqtt_loggers(*logger=None*)

Parameters
logger (*Logger | LoggerAdapter | None*) –

generate_event(*event*)

Parameters
event (*EventT*) –

Return type
Ok[bool] | *Err*[BaseException]

get_message_times(*link_name*)

Parameters
link_name (*str*) –

Return type
LinkMessageTimes

link(*name*)

Return type
LinkState | None

link_names()

Return type
list[str]

link_state(*name*)

Return type
StateName | None

log_subscriptions(*tag=''*)

mqtt_client_wrapper(*client_name*)

Parameters
client_name (*str*) –

Return type
MQTTClientWrapper

mqtt_clients()

Return type
MQTTClients

num_acks(*link_name*)

Parameters
link_name (*str*) –

Return type
int

property num_pending: int

property num_reupload_pending: int

property num_reuploaded_unacked: int

property primary_peer_client: str

process_ack(*link_name*, *message_id*)

Parameters

- **link_name** (*str*) –
- **message_id** (*str*) –

process_ack_timeout(*wait_info*)

Parameters
wait_info (AckWaitInfo) –

Return type
Ok[LinkManagerTransition] | *Err*[BaseException]

process_mqtt_connect_fail(*message*)

Parameters
message (*Message[MQTTConnectFailPayload]*) –

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

process_mqtt_connected(*message*)

Parameters
message (*Message[MQTTConnectPayload]*) –

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

process_mqtt_disconnected(*message*)

Parameters
message (*Message[MQTTDisconnectPayload]*) –

Return type
Ok[LinkManagerTransition] | *Err*[InvalidCommStateInput]

process_mqtt_message(*message*)

Parameters
message (*Message[MQTTReceiptPayload]*) –

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

process_mqtt_suback(*message*)

Parameters
message (*Message[MQTTSubackPayload]*) –

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

publication_name: str

publish_message(*client*, *message*, *qos=0*, *context=None*)

Parameters

- **message** (*Message*) –
- **qos** (*int*) –
- **context** (*Any*) –

Return type
MQTTMessageInfo

publish_upstream(*payload*, *qos=QOS.AtMostOnce*, ***message_args*)

Parameters

- **qos** (QOS) –
- **message_args** (*Any*) –

Return type
MQTTMessageInfo

reuploading()

Return type
bool

send_ack(*link_name*, *message*)

Parameters

- **link_name** (*str*) –
- **message** (*Message[Any]*) –

Return type
None

async send_ping(*link_name*)

Parameters
link_name (*str*) –

start(*loop*, *async_queue*)

Parameters

- **loop** (*AbstractEventLoop*) –
- **async_queue** (*Queue*) –

Return type
None

start_ping_tasks()

Return type
list[*Task*]

stop()

Return type
Ok[bool] | *Err*[Problems]

stopped(*name*)

Parameters
name (*str*) –

Return type
bool

subscribe(*client*, *topic*, *qos*)

Parameters

- **client** (*str*) –
- **topic** (*str*) –
- **qos** (*int*) –

Return type
Tuple[int, int | None]

subscribed(*link_name*)

Parameters
link_name (*str*) –

Return type
bool

update_recv_time(*link_name*)

Parameters
link_name (*str*) –

Return type
None

property upstream_client: str

class gwproactor.links.**LinkManagerTransition**(*link_name: str = ''*, *transition_name: gwproactor.links.link_state.TransitionName = <TransitionName.none: 'none'>*, *old_state: gwproactor.links.link_state.StateName = <StateName.not_started: 'not_started'>*, *new_state: gwproactor.links.link_state.StateName = <StateName.not_started: 'not_started'>*, *canceled_acks: list[gwproactor.links.acks.AckWaitInfo] = <factory>*)

Parameters

- **link_name** (*str*) –
- **transition_name** (TransitionName) –
- **old_state** (StateName) –
- **new_state** (StateName) –
- **canceled_acks** (*list[AckWaitInfo]*) –

canceled_acks: list[*AckWaitInfo*]

class gwproactor.links.**LinkMessageTimes**(*last_send: float = <factory>*, *last_recv: float = <factory>*)

Parameters

- **last_send** (*float*) –
- **last_recv** (*float*) –

get_str(*link_poll_seconds=60.0*, *relative=True*)

Parameters

- **link_poll_seconds** (*float*) –
- **relative** (*bool*) –

Return type
str

last_recv: float

last_send: float

next_ping_second(*link_poll_seconds*)

Parameters
link_poll_seconds (*float*) –

Return type
float

seconds_until_next_ping(*link_poll_seconds*)

Parameters
link_poll_seconds (*float*) –

Return type
float

time_to_send_ping(*link_poll_seconds*)

Parameters
link_poll_seconds (*float*) –

Return type
bool

class gwproactor.links.**LinkState**(*name*, *curr_state=StateName.not_started*)

Parameters

- **name** (*str*) –
- **curr_state** (*State*) –

_handle(*result*)

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

active()

active_for_recv()

active_for_send()

curr_state: State

in_state(*state*)

Parameters
state (StateName) –

Return type
bool

name: str

process_ack_timeout()

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

process_mqtt_connect_fail()

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

process_mqtt_connected()

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

process_mqtt_disconnected()

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

process_mqtt_message()

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

process_mqtt_suback(*num_pending_subscriptions*)

Parameters
num_pending_subscriptions (*int*) –

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

start()

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

property state: *StateName*

states: dict[*StateName*, State]

stop()

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

class gwproactor.links.**LinkStates**(*names=None*)

Parameters
names (*Sequence[str] | None*) –

_links: dict[str, *LinkState*]

add(*name*, *state=StateName.not_started*)

Parameters

- **name** (*str*) –
- **state** (StateName) –

Return type
LinkState

link(*name*)

Parameters
name (*str*) –

Return type
LinkState | None

link_names()

Return type
list[str]

link_state(*name*)

Parameters
name (*str*) –

Return type
StateName | None

process_ack_timeout(*name*)

Parameters
name (*str*) –

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

process_mqtt_connect_fail(*message*)

Parameters
message (*Message[MQTTConnectFailPayload]*) –

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

process_mqtt_connected(*message*)

Parameters
message (*Message[MQTTConnectPayload]*) –

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

process_mqtt_disconnected(*message*)

Parameters
message (*Message[MQTTDisconnectPayload]*) –

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

process_mqtt_message(*message*)

Parameters
message (*Message[MQTTReceiptPayload]*) –

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

process_mqtt_suback(*name*, *num_pending_subscriptions*)

Parameters

- **name** (*str*) –
- **num_pending_subscriptions** (*int*) –

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

start(*name*)

Parameters
name (*str*) –

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

start_all()

Return type
Ok[bool] | *Err*[*Sequence*[BaseException]]

stop(*name*)

Parameters
name (*str*) –

Return type
Ok[Transition] | *Err*[InvalidCommStateInput]

stopped(*name*)

Parameters
name (*str*) –

Return type
bool

class gwproactor.links.**MQTTClientWrapper**(*name*, *client_config*, *receive_queue*)

Parameters

- **name** (*str*) –
- **client_config** (*MQTTClient*) –
- **receive_queue** (AsyncQueueWriter) –

_client: Client

_client_config: MQTTClient

_client_thread()

_name: str

_pending_subacks: Dict[int, List[str]]

_pending_subscriptions: Set[str]

_receive_queue: *AsyncQueueWriter*

_stop_requested: bool

_subscriptions: Dict[str, int]

connected()

Return type
bool

disable_logger()

enable_logger(*logger=None*)

Parameters
logger (*Logger | LoggerAdapter | None*) –

handle_suback(*suback*)

Parameters
suback (*MQTTSubackPayload*) –

Return type
int

num_pending_subscriptions()

Return type
int

num_subscriptions()

Return type
int

on_connect(_, *userdata*, *flags*, *rc*)

on_connect_fail(_, *userdata*)

on_disconnect(_, *userdata*, *rc*)

on_message(_, *userdata*, *message*)

on_subscribe(_, *userdata*, *mid*, *granted_qos*)

publish(*topic*, *payload*, *qos*)

Parameters

- **topic** (*str*) –
- **payload** (*bytes*) –
- **qos** (*int*) –

Return type
MQTTMessageInfo

start()

stop()

subscribe(*topic*, *qos*)

Parameters

- **topic** (*str*) –
- **qos** (*int*) –

Return type
Tuple[int, int | None]

subscribe_all()

Return type
Tuple[int, int | None]

subscribed()

Return type
bool

subscription_items()

Return type
list[*Tuple*[str, int]]

unsubscribe(*topic*)

Parameters
topic (*str*) –

Return type
Tuple[int, int | None]

class gwproactor.links.**MQTTClients**

_send_queue: *AsyncQueueWriter*

add_client(*name*, *client_config*, *upstream=False*, *primary_peer=False*)

Parameters

- **name** (*str*) –
- **client_config** (*MQTTClient*) –
- **upstream** (*bool*) –
- **primary_peer** (*bool*) –

client_wrapper(*client*)

Parameters
client (*str*) –

Return type
MQTTClientWrapper

clients: Dict[str, *MQTTClientWrapper*]

connected(*client*)

Parameters
client (*str*) –

Return type
bool

disable_loggers()

enable_loggers(*logger=None*)

Parameters
logger (*Logger | LoggerAdapter | None*) –

handle_suback(*suback*)

Parameters
suback (*MQTTSubackPayload*) –

Return type
int

num_pending_subscriptions(*client*)

Parameters
client (*str*) –

Return type
int

num_subscriptions(*client*)

Parameters
client (*str*) –

Return type
int

primary_peer()

Return type
MQTTClientWrapper

primary_peer_client: str = ''

publish(*client*, *topic*, *payload*, *qos*)

Parameters

- **client** (*str*) –
- **topic** (*str*) –
- **payload** (*bytes*) –
- **qos** (*int*) –

Return type
MQTTMessageInfo

start(*loop*, *async_queue*)

Parameters

- **loop** (*AbstractEventLoop*) –
- **async_queue** (*Queue*) –

stop()

subscribe(*client*, *topic*, *qos*)

Parameters

- **client** (*str*) –
- **topic** (*str*) –
- **qos** (*int*) –

Return type
Tuple[int, int | None]

subscribe_all(*client*)

Parameters
client (*str*) –

Return type
Tuple[int, int | None]

subscribed(*client*)

Parameters
client (*str*) –

Return type
bool

unsubscribe(*client*, *topic*)

Parameters

- **client** (*str*) –
- **topic** (*str*) –

Return type
Tuple[int, int | None]

upstream()

Return type
MQTTClientWrapper

upstream_client: str = ''

class gwproactor.links.**MessageTimes**

_links: dict[str, *LinkMessageTimes*]

add_link(*name*)

Parameters
name (*str*) –

Return type
None

get_copy(*link_name*)

Parameters
link_name (*str*) –

Return type
LinkMessageTimes

link_names()

Return type
list[str]

update_recv(*link_name*, *now=None*)

Parameters

- **link_name** (*str*) –
- **now** (*float | None*) –

Return type
None

update_send(*link_name*, *now=None*)

Parameters

- **link_name** (*str*) –
- **now** (*float | None*) –

Return type
None

class gwproactor.links.**QOS**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

AtLeastOnce = 1

AtMostOnce = 0

ExactlyOnce = 2

class gwproactor.links.**Reuploads**(*event_persister*, *logger*, *num_initial_events=5*)

Parameters

- **event_persister** (*PersisterInterface*) –
- **logger** (ProactorLogger) –
- **num_initial_events** (*int*) –

NUM_INITIAL_EVENTS: int = 5

_event_persister: PersisterInterface

_logger: *ProactorLogger*

_num_initial_events: int

_reupload_pending: dict[str, None]

_reuploaded_unacked: dict[str, None]

clear()

Return type
None

property num_reupload_pending: int

property num_reuploaded_unacked: int

process_ack_for_reupload(*message_id*)

Parameters
message_id (*str*) –

Return type
list[str]

reuploading()

Return type
bool

start_reupload()

Return type
list[str]

exception gwproactor.links.**RuntimeLinkStateError**(*name=''*, *current_state=StateName.none*, *transition=TransitionName.none*, ***, *msg=''*)

Parameters

- **name** (*str*) –
- **current_state** (StateName) –
- **transition** (TransitionName) –
- **msg** (*str*) –

class gwproactor.links.**StateName**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

active = 'active'

awaiting_peer = 'awaiting_peer'

awaiting_setup = 'awaiting_setup'

awaiting_setup_and_peer = 'awaiting_setup_and_peer'

connecting = 'connecting'

none = 'none'

not_started = 'not_started'

stopped = 'stopped'

class gwproactor.links.**Subscription**(*Topic*, *Qos*)

Parameters

- **Topic** (*str*) –
- **Qos** (QOS) –

Qos: *QOS*

Alias for field number 1

Topic: str

Alias for field number 0

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('Topic', 'Qos')

classmethod _make(*iterable*)

Make a new Subscription object from a sequence or iterable

_replace(***kwds*)

Return a new Subscription object replacing specified fields with new values

class gwproactor.links.**TimerManagerInterface**

Simple interface to infrastructure which can start timers, run callbacks on timer completion, and cancel timers.

_abc_impl = <_abc._abc_data object>

abstract cancel_timer(*timer_handle*)

Cancel callback associated with _timer_handle_.

Note that callback might still run after this call returns.

Parameters
timer_handle (*Any*) – The value returned by start_timer()

Return type
None

abstract start_timer(*delay_seconds*, *callback*)

Start a timer. Implementation is expected to call _callback_ after approximately _delay_sceonds_.

The execution context (e.g. the thread) of the callback must be specified by the implemntation.

The callback must have sufficient context available to it do its work as well as to detect if it is no longer relevant. Note a callback might run after cancelation if the callack was already "in-flight" at time of cancellation and it is up to the callback to tolerate this situation.

Parameters

- **delay_seconds** (*float*) – The approximate delay before the callback is called.
- **callback** (*Callable[[], None]*) – The function called after delay_seconds.

Returns
A timer handle which can be passed to _cancel_timer()_ to cancel the callback.

Return type
Any

class gwproactor.links.**Transition**(*link_name: str = ''*, *transition_name: gwproactor.links.link_state.TransitionName = <TransitionName.none: 'none'>*, *old_state: gwproactor.links.link_state.StateName = <StateName.not_started: 'not_started'>*, *new_state: gwproactor.links.link_state.StateName = <StateName.not_started: 'not_started'>*)

Parameters

- **link_name** (*str*) –
- **transition_name** (TransitionName) –
- **old_state** (StateName) –
- **new_state** (StateName) –

activated()

active()

deactivated()

link_name: str = ''

new_state: *StateName* = 'not_started'

old_state: *StateName* = 'not_started'

recv_activated()

Return type
bool

recv_deactivated()

Return type
bool

send_activated()

Return type
bool

send_deactivated()

Return type
bool

send_is_active()

Return type
bool

transition_name: *TransitionName* = 'none'

class gwproactor.links.**TransitionName**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

message_from_peer = 'message_from_peer'

mqtt_connect_failed = 'mqtt_connect_failed'

mqtt_connected = 'mqtt_connected'

mqtt_disconnected = 'mqtt_disconnected'

mqtt_suback = 'mqtt_suback'

none = 'none'

response_timeout = 'response_timeout'

start_called = 'start_called'

stop_called = 'stop_called'

7.2.3 gwproactor_test

Development package used to test classes inheriting from *gwproactor.Proactor*

7.3 Contributor Guide

Thank you for your interest in improving this project. This project is open-source under the MIT license and welcomes contributions in the form of bug reports, feature requests, and pull requests.

Here is a list of important resources for contributors:

- Source Code
- Documentation
- Issue Tracker
- *Code of Conduct*

7.3.1 How to report a bug

Report bugs on the Issue Tracker.

When filing an issue, make sure to answer these questions:

- Which operating system and Python version are you using?
- Which version of this project are you using?
- What did you do?
- What did you expect to see?
- What did you see instead?

The best way to get your bug fixed is to provide a test case, and/or steps to reproduce the issue.

7.3.2 How to request a feature

Request features on the Issue Tracker.

7.3.3 How to set up your development environment

You need Python 3.7+ and the following tools:

- Poetry
- Nox
- nox-poetry

Install the package with development requirements:

```
$ poetry install --all-extras
```

You can now run an interactive Python session, or the command-line interface:

```
$ poetry run python
$ poetry run gridworks-proactor
```

7.3.4 How to test the project

Run the full test suite:

```
$ nox
```

List the available Nox sessions:

```
$ nox --list-sessions
```

You can also run a specific Nox session. For example, invoke the unit test suite like this:

```
$ nox --session=tests
```

Unit tests are located in the *tests* directory, and are written using the pytest testing framework.

7.3.5 How to submit changes

Open a pull request to submit changes to this project.

Your pull request needs to meet the following guidelines for acceptance:

- The Nox test suite must pass without errors and warnings.
- Include unit tests. This project maintains 100% code coverage.
- If your changes add functionality, update the documentation accordingly.

Feel free to submit early, though—we can always iterate on this.

To run linting and code formatting checks before committing your change, you can install pre-commit as a Git hook by running the following command:

```
$ nox --session=pre-commit -- install
```

It is recommended to open an issue before starting work on anything. This will allow a chance to talk it over with the owners and validate your approach.

7.4 GridWorks Energy Consulting Code of Conduct

7.4.1 Basic Truth

All humans are worthy.

7.4.2 Scope

This Code of Conduct applies to moderation of comments, issues and commits within this repository to support its alignment to the above basic truth.

7.4.3 Enforcement Responsibilities

GridWorks Energy Consulting LLC (gridworks@gridworks-consulting.com) owns and administers this repository, and is ultimately responsible for enforcement of standards of behavior. They are responsible for merges to dev and main branches, and maintain the right and responsibility to remove, edit, or reject comments, commits, code, docuentation edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

If you read something in this repo that you want GridWorks to consider moderating, please send an email to them at gridworks@gridworks-consulting.com. All complaints will be reviewed and investigated, and GridWorks will respect the privacy and security of the reporter of any incident.

7.4.4 What not to add to this repo

Ways to trigger GridWorks moderation enforcement:

- Publish others' private information, such as a physical or email address, without their explicit permission
- Use of sexualized language or imagery, or make sexual advances
- Troll

7.4.5 Suggestions

- Empathize
- Recognize you are worthy of contributing, and do so in the face of confusion and doubt; you can help clarify things for everyone
- Be interested in differing opinions, viewpoints, and experiences
- Give and accept constructive feedback
- Accept responsibility for your mistakes and learn from them
- Recognize everybody makes mistakes, and forgive
- Focus on the highest good for all

7.4.6 Enforcement Escalation

1. Correction

A private, written request from GridWorks to change or edit a comment, commit, or issue.

2. Warning

With a warning, GridWorks may remove your comments, commits or issues. They may also freeze a conversation.

3. Temporary Ban

A temporary ban from any sort of interaction or public communication within the repository for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

A permanent ban from any sort of interaction within the repository.

7.4.7 Attribution

This Code of Conduct is loosely adapted from the Contributor Covenant, version 2.1, available at https://www.contributor-covenant.org/version/2/1/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

7.5 License

PYTHON MODULE INDEX

g

INDEX

Symbols

A

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

W