KATCP Documentation

Release 0.0+unknown.202105130806

Simon Cross

May 13, 2021

Contents

KATCP is a simple ASCII communication protocol layered on top of TCP/IP.

It is developed as a part of the Karoo Array Telescope (KAT) project and used at KAT for the monitoring and control of hardware devices.

The protocol specification `NRF-KAT7-6.0-IFCE-002-Rev5-1.pdf` is maintained as an internal memo. The latest version is *Rev5.1*. The specification source is hosted here: https://github.com/ska-sa/katcp-guidelines

CHAPTER 1

Contents

1.1 Release Notes

1.1.1 0.9.1

- Fix issues in KATCPReply *__repr__* in py3

1.1.2 0.9.0

- Add asyncio compatible ioloop to ioloop manager.

1.1.3 0.8.0

- Added bulk sensor sampling feature.

1.1.4 0.7.2

- Support for handling generator expressions in `Discrete` type.
- Fix handling of strings and bytes in `get_sensor` in testutils.
- Allow strings or bytes for `assert_request_fails` and `test_assert_request_succeeds` function arguments.
- Handle `str` type correctly ('easier') in `testutils.get_sensor` for python 2 and python 3.
- Allow bytes and strings in `test_sensor_list` comparison of sensors.
- Correct handling of floats `test_sensor_list`.
- black formatting on certain test files.

1.1.5 0.7.1

- All params in `future_get_sensor` are now cast to byte strings.
- Added tests to `test_fake_clients.py` and `test_inspecting_client.py`.
- Ensure `testutils` method casts expected requests to byte strings.

1.1.6 0.7.0

- Added Python 3 compatibility.

See also `CHANGELOG.md` for more details on changes.

Important changes for Python 3 compatibility

General notes

The package is now compatible with both Python 2 and 3. The goals of the migration were:

- Do not change the public API.
- Do not break existing functionality for Python 2.
- Ease migration of packages using katcp to Python 3.

Despite these goals, some of the stricter type checking that has been added may force minor updates in existing code. E.g., using integer for the options of a discrete sensor is no longer allowed.

Asynchronous code is still using tornado in the same Python 2 way. The new Python 3.5 `async` and `await` keywords are not used. The tornado version is also pinned to older versions that support both Python 2 and 3. The 5.x versions also support Python 2, but they are avoided as some significant changes result in test failures.

The Python `future` package was used for the compatibility layer. The use of the `newstr` and `newbytes` compatibility types was avoided, to reduce confusion. I.e., `from builtins import str, bytes` is not done.

Docstrings

In docstrings the interpretation of parameter and return types described as "str" has changed slightly. In Python 2 the `str` type is a byte string, while in Python 3, `str` is a unicode string. The `str` type is referred to as the "native" string type. In code, native literal strings would have no prefix, for example: `"native string"`, as opposed to explicit byte strings, `b"byte string"`, and explicit unicode strings, `u"unicode string"`. In the docstrings "bytes" means a byte string is expected (or returned), "str" means a native string, and "str or bytes" means either type.

Changes to types

As part of the Python 3 compatibility update, note the following:

- *`katcp.Message`*. - `arguments` and `mid` attributes will be forced to byte strings in all Python versions. This is to match what is sent on the wire (serialised byte stream). - `name`: is expected to be a native string. - `repr()`: the result will differ slightly in Python 3 - the arguments will be shown as quoted byte strings. E.g., Python 2: `"<Message reply ok (123, zzz)>"`, vs. Python 3: `"<Message reply ok (b'123', b'zzz')>"`. In all versions, arguments longer than 1000 characters are now truncated.

- *katcp.Sensor*. - name, description, units, params (for discrete sensors): __init__ can take byte strings or native strings, but attributes will be coerced to native strings. - set_formatted, parse_value: only accept byte strings (stricter checking). - The float and strict_timestamp sensor values are now encoded using repr() instead of "%.15g". This means that more significant digits are transmitted on the wire (16 to 17, instead of 15), and the client will be able to reconstruct the exact some floating point value.

Non-ASCII and UTF-8

Prior to these changes, all strings were byte strings, so there was no encoding required. Arbitrary bytes could be used for message parameters and string sensor values. After these changes, strings sensors and Str types are considered "text". In Python 3, UTF-8 encoding will be used when changing between byte strings and unicode strings for "text". This has the following effects:

- *katcp.Message* - the arguments are always using byte strings, so arbitrary bytes can still be sent and received using this class directly.
- *katcp.Sensor* - Values for string and discrete sensor types cannot be arbitrary byte strings in Python 3 - they need to be UTF-8 compatible.
- kattypes.Str, kattypes.Discrete, kattypes.DiscreteMulti - These types is still used in request and reply decorators. - For sending messages, they accept any type of object, but UTF-8 encoding is used if values are not already byte strings. - When decoding received messages, "native" strings are returned.

Keep in mind that a Python 2 server may be communicating with a Python 3 client, so sticking to ASCII is safest. If you are sure both client and server are on Python 3 (or understand the encoding the same), then UTF-8 could be used. That is also the encoding option used by the aiokatcp package.

Performance degradation

Adding the compatibility results in more checks and conversions. From some basic benchmarking, there appears to be up to 20% performance degradation when instantiating message objects.

Benchmark, in ipython:

```
import random, katcp

args_groups = []
for i in range(1000):
    args_groups.append((random.randint(0, 1) == 1,
                        random.randint(0, 1000),
                        random.random(),
                        str(random.random())))

def benchmark():
    for args in args_groups:
        tx_msg = katcp.Message.reply('foo', *args)
        serialised = bytes(tx_msg)
        parser = katcp.MessageParser()
        rx_msg = parser.parse(serialised)
        assert tx_msg == rx_msg


%timeit benchmark()
```

- Old Py2: 10 loops, best of 3: 23.4 ms per loop

- New Py2: 10 loops, best of 3: 29.9 ms per loop
- New Py3: 25.1 ms ± 86.8 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

1.1.7 0.6.4

- Fix some client memory leaks, and add *until_stopped* methods.
- Increase server MAX_QUEUE_SIZE to handle more clients.
- Use correct ioloop for client AsyncEvent objects.

See also `CHANGELOG.md` for more details on changes.

Important API changes

Stopping KATCP clients

When stopping KATCP client classes that use a *managed* ioloop (i.e., create their own in a new thread), the traditional semantics are to call `stop()` followed by `join()` from another thread. This is unchanged. In the case of an *unmanaged* ioloop (i.e., an existing ioloop instance is provided to the client), we typically stop from the same thread, and calling `join()` does nothing. For the case of *unmanaged* ioloops, a new method, `until_stopped()`, has been added. It returns a future that resolves when the client has stopped. The caller can `yield` on this future to be sure that the client has completed all its coroutines. Using this new method is not required. If the ioloop will keep running, the stopped client's coroutines will eventually exit. However, it is useful in some cases, e.g., to verify correct clean up in unit tests.

The new method is available on *katcp.DeviceClient* and derived classes, on *katcp.inspecting_client.InspectingClientAsync*, and on the high-level clients *katcp.KATCPClientResource* and *katcp.KATCPClientResourceContainer*.

An additional change is that the inspecting client now sends a state update (indicating that it is disconnected and not synced) when stopping. This means high-level clients that were waiting on `until_not_synced` when the client was stopped will now be notified. Previously, this was not the case.

1.1.8 0.6.3

- Put docs on readthedocs.
- Better error handling for messages with non-ASCII characters (invalid).
- Increase container sync time to better support large containers.
- Limit tornado version to <5.
- Allow sampling strategy to be removed from cache.
- Improve error messages for DeviceMetaClass assertions.
- Increase server's message queue length handle more simultaneous client connections.
- Improve Jenkins pipeline configuration.
- Add information on how to contribute to the project.

See also `CHANGELOG.md` for more details on changes.

1.1.9 0.6.2

- Various bug fixes
- Docstring and code style improvements
- Bumped the tornado dependency to at least 4.3
- Added the ability to let ClientGroup wait for a quorum of clients
- Added default request-timeout-hint implementation to server.py
- Moved IOLoopThreadWrapper to ioloop_manager.py, a more sensible location
- Added a random-exponential retry backoff process

See also `CHANGELOG.md` for more details on changes.

1.1.10 0.6.1

- Various bug fixes
- Improvements to testing utilities
- Improvements to various docstrings
- Use *katversion* to determine version string on install
- Better dependency management using setup.py with *setuptools*
- Fixed a memory leak when using KATCPResourceContainer

See also `CHANGELOG.md` for more details on changes.

1.1.11 0.6.0

- Major change: Use the tornado event loop and async socket routines.

See also `CHANGELOG.md` for more details on changes.

Important API changes

Tornado based event loop(s)

While the networking stack and event loops have been re-implemented using Tornado, this change should be largely invisible to existing users of the library. All client and server classes now expose an *ioloop* attribute that is the `tornado.ioloop.IOLoop` instance being used. Unless new server or client classes are used or default settings are changed, the thread-safety and concurrency semantics of 0.5.x versions should be retained. User code that made use of non-public interfaces may run into trouble.

High level auto-inspecting KATCP client APIs added

The high level client API inspects a KATCP device server and present requests as method calls and sensors as objects. See *Using the high-level client API*.

Sensor observer API

The *katcp.Sensor* sensor observer API has been changed to pass the sensor reading in the *observer.update()* callback, preventing potential lost updates due to race conditions. This is a backwards incompatible change. Whereas before observers were called as *observer.update(sensor)*, they are now called as *observer.update(sensor, reading)*, where *reading* is an instance of `katcp.core.Reading`.

Sample Strategy callback API

Sensor strategies now call back with the sensor object and raw Python datatype values rather than the sensor name and KATCP formatted values. The sensor classes have also grown a *katcp.Sensor.format_reading()* method that can be used to do KATCP-version specific formatting of the sensor reading.

1.1.12 0.5.5

- Various cleanups (logging, docstrings, base request set, minor refactoring)
- Improvements to testing utilities
- Convenience utility functions in *katcp.version*, *katcp.client*, *katcp.testutils*.

1.1.13 0.5.4

- Change event-rate strategy to always send an update if the sensor has changed and shortest-period has passed.
- Add differential-rate strategy.

1.1.14 0.5.3

Add `convert_seconds()` method to katcp client classes that converts seconds into the device timestamp format.

1.1.15 0.5.2

Fix memory leak in sample reactor, other minor fixes.

1.1.16 0.5.1

Minor bugfixes and stability improvements

1.1.17 0.5.0

First stable release supporting (a subset of) KATCP v5. No updates apart from documentation since 0.5.0a0; please refer to the 0.5.0a release notes below.

1.1.18 0.5.0a0

First alpha release supporting (a subset of) KATCP v5. The KATCP v5 spec brings a number of backward incompatible changes, and hence requires care. This library implements support for both KATCP v5 and for the older dialect. Some API changes have also been made, mainly in aid of fool-proof support of the Message ID feature of KATCP v5. The changes do, however, also eliminate a category of potential bugs for older versions of the spec.

Important API changes

CallbackClient.request()

Renamed `request()` to *`callback_request()`* to be more consistent with superclass API.

Sending replies and informs in server request handlers

The function signature used for request handler methods in previous versions of this library were *request_requestname(self, sock, msg)*, where *sock* is a raw python socket object and msg is a katcp `Message` object. The *sock* object was never used directly by the request handler, but was passed to methods on the server to send inform or reply messages.

Before:

```
class MyServer(DeviceServer):
    def request_echo(self, sock, msg):
        self.inform(sock, Message.inform('echo', len(msg.arguments)))
        return Message.reply('echo', 'ok', *msg.arguments)
```

The old method requires the name of the request to be repeated several times, inviting error and cluttering code. The user is also required to instantiate katcp `Message` object each time a reply is made. The new method passes a request-bound connection object that knows to what request it is replying, and that automatically constructs `Message` objects.

Now:

```
class MyServer(DeviceServer):
    def request_echo(self, req, msg):
        req.inform(len(msg.arguments)))
        return req.make_reply('ok', *msg.arguments)
```

A `req.reply()` method with the same signature as `req.make_reply()` is also available for asyncronous reply handlers, and `req.reply_with_message()` which takes a `Message` instance rather than message arguments. These methods replace the use of `DeviceServer.reply()`.

The request object also contains the katcp request `Message` object (*req.msg*), and the equivalent of a socket object (*req.client_connection*). See the next section for a description of *client_connection*.

Using the server methods with a *req* object in place of *sock* will still work as before, but will log deprecation warnings.

Connection abstraction

Previously, the server classes internally used each connection's low-level *sock* object as an identifier for the connection. In the interest of abstracting out the transport backend, the *sock* object has been replaced by a

`ClientConnectionTCP` object. This object is passed to all server handler functions (apart from request handlers) instead of the *sock* object. The connection object be used in the same places where *sock* was previously used. It also defines `inform()`, `reply_inform()` and `reply()` methods for sending `Message` objects to a client.

Backwards incompatible KATCP V5 changes

Timestamps

Excerpted from `NRF-KAT7-6.0-IFCE-002-Rev5.pdf`:

> All core messages involving time (i.e. timestamp or period specifications) have changed from using milliseconds to seconds. This provides consistency with SI units. Note also that from version five timestamps should always be specified in UTC time.

Message Identifiers (mid)

Excerpted from `NRF-KAT7-6.0-IFCE-002-Rev5.pdf`:

> Message identifiers were introduced in version 5 of the protocol to allow replies to be uniquely associated with a particular request. If a client sends a request with a message identifier the server must include the same identifier in the reply. Message identifiers are limited to integers in the range 1 to 231 1 inclusive. It is the client's job to construct suitable identifiers – a server should not assume that these are unique. Clients that need to determine whether a server supports message identifiers should examine the #version-connect message returned by the server when the client connects (see Section 4). If no #version-connect message is received the client may assume message identifiers are not supported.

also:

> If the request contained a message id each inform that forms part of the response should be marked with the original message id.

Support for message IDs is optional. A properly implemented server should never use mids in replies unless the client request has an mid. Similarly, a client should be able to detect whether a server supports MIDs by checking the *#version-connect* informs sent by the server, or by doing a *!version-list* request. Furthermore, a KATCP v5 server should never send *#build-state* or *#version* informs.

Server KATCP Version Auto-detection

The *DeviceClient* client uses the presence of *#build-state* or *#version* informs as a heuristic to detect pre-v5 servers, and the presence of *#version-connect* informs to detect v5+ servers. If mixed messages are received the client gives up auto-detection and disconnects. In this case *preset_protocol_flags()* can be used to configure the client before calling *start()*.

Level of KATCP support in this release

This release implements the majority of the KATCP v5 spec; excluded parts are:

- Support for optional warning/error range meta-information on sensors.
- Differential-rate sensor strategy.

1.2 Core API

1.2.1 Client

Two different clients are provided: the *BlockingClient* for synchronous communication with a server and the *CallbackClient* for asynchronous communication. Both clients raise *KatcpClientError* when exceptions occur.

The *DeviceClient* base class is provided as a foundation for those wishing to implement their own clients.

BlockingClient

class katcp.**BlockingClient**(*host*, *port*, *tb_limit=20*, *timeout=5.0*, *logger=<logging.Logger object>*, *auto_reconnect=True*)

Methods

BlockingClient.blocking_request(msg[, ...])	Send a request message and wait for its reply.
BlockingClient.callback_request(msg[, ...])	Send a request message.
BlockingClient.convert_seconds(time_seconds)	Convert a time in seconds to the device timestamp units.
BlockingClient.disconnect()	Force client connection to close, reconnect if auto-connect set.
BlockingClient.enable_thread_safety()	Enable thread-safety features.
BlockingClient.future_request(msg[, ...])	Send a request message, with future replies.
BlockingClient.handle_inform(msg)	Handle inform messages related to any current requests.
BlockingClient.handle_message(msg)	Handle a message from the server.
BlockingClient.handle_reply(msg)	Handle a reply message related to the current request.
BlockingClient.handle_request(msg)	Dispatch a request message to the appropriate method.
BlockingClient.inform_build_state(msg)	Handle katcp v4 and below build-state inform.
BlockingClient.inform_version(msg)	Handle katcp v4 and below version inform.
BlockingClient.inform_version_connect(msg)	Process a #version-connect message.
BlockingClient.is_connected()	Check if the socket is currently connected.
BlockingClient.join([timeout])	Rejoin the client thread.
BlockingClient.next()	
BlockingClient.notify_connected(connected)	Event handler that is called whenever the connection status changes.
BlockingClient.preset_protocol_flags(...)	Preset server protocol flags.
BlockingClient.request(msg[, use_mid])	Send a request message, with automatic message ID assignment.

Continued on next page

Table 1 – continued from previous page

BlockingClient.running()	Whether the client is running.
BlockingClient.send_message(msg)	Send any kind of message.
BlockingClient.send_request(msg)	Send a request message.
BlockingClient.setDaemon(daemonic)	Set daemonic state of the managed ioloop thread to True / False
BlockingClient.set_ioloop([ioloop])	Set the tornado.ioloop.IOLoop instance to use.
BlockingClient.start([timeout])	Start the client in a new thread.
BlockingClient.stop(*args, **kwargs)	Stop a running client.
BlockingClient.unhandled_inform(msg)	Fallback method for inform messages without a registered handler.
BlockingClient.unhandled_reply(msg)	Fallback method for reply messages without a registered handler.
BlockingClient.unhandled_request(msg)	Fallback method for requests without a registered handler.
BlockingClient.until_connected(**kwargs)	Return future that resolves when the client is connected.
BlockingClient.until_protocol(**kwargs)	Return future that resolves after receipt of katcp protocol info.
BlockingClient.until_running([timeout])	Return future that resolves when the client is running.
BlockingClient.until_stopped([timeout])	Return future that resolves when the client has stopped.
BlockingClient.wait_connected([timeout])	Wait until the client is connected.
BlockingClient.wait_disconnected([timeout])	Wait until the client is disconnected.
BlockingClient.wait_protocol([timeout])	Wait until katcp protocol information has been received from server.
BlockingClient.wait_running([timeout])	Wait until the client is running.

bind_address
(host, port) where the client is connecting

blocking_request(*msg*, *timeout=None*, *use_mid=None*)
Send a request message and wait for its reply.

Parameters **msg** : Message object

The request Message to send.

timeout : float in seconds

How long to wait for a reply. The default is the the timeout set when creating the AsyncClient.

use_mid : boolean, optional

Whether to use message IDs. Default is to use message IDs if the server supports them.

Returns **reply** : Message object

The reply message received.

informs : list of Message objects

A list of the inform messages received.

callback_request(*msg*, *reply_cb=None*, *inform_cb=None*, *user_data=None*, *timeout=None*, *use_mid=None*)

Send a request message.

Parameters **msg** : Message object

The request message to send.

reply_cb : function

The reply callback with signature reply_cb(msg) or reply_cb(msg, *user_data)

inform_cb : function

The inform callback with signature inform_cb(msg) or inform_cb(msg, *user_data)

user_data : tuple

Optional user data to send to the reply and inform callbacks.

timeout : float in seconds

How long to wait for a reply. The default is the the timeout set when creating the AsyncClient.

use_mid : boolean, optional

Whether to use message IDs. Default is to use message IDs if the server supports them.

convert_seconds(*time_seconds*)

Convert a time in seconds to the device timestamp units.

KATCP v4 and earlier, specified all timestamps in milliseconds. Since KATCP v5, all timestamps are in seconds. If the device KATCP version has been detected, this method converts a value in seconds to the appropriate (seconds or milliseconds) quantity. For version smaller than V4, the time value will be truncated to the nearest millisecond.

disconnect()

Force client connection to close, reconnect if auto-connect set.

enable_thread_safety()

Enable thread-safety features.

Must be called before start().

future_request(*msg*, *timeout=None*, *use_mid=None*)

Send a request message, with future replies.

Parameters **msg** : Message object

The request Message to send.

timeout : float in seconds

How long to wait for a reply. The default is the the timeout set when creating the AsyncClient.

use_mid : boolean, optional

Whether to use message IDs. Default is to use message IDs if the server supports them.

Returns **A tornado.concurrent.Future that resolves with:** :

reply : Message object

The reply message received.

informs : list of Message objects

A list of the inform messages received.

handle_inform(*msg*)
Handle inform messages related to any current requests.

Inform messages not related to the current request go up to the base class method.

Parameters **msg** : Message object

The inform message to dispatch.

handle_message(*msg*)
Handle a message from the server.

Parameters **msg** : Message object

The Message to dispatch to the handler methods.

handle_reply(*msg*)
Handle a reply message related to the current request.

Reply messages not related to the current request go up to the base class method.

Parameters **msg** : Message object

The reply message to dispatch.

handle_request(*msg*)
Dispatch a request message to the appropriate method.

Parameters **msg** : Message object

The request message to dispatch.

inform_build_state(*msg*)
Handle katcp v4 and below build-state inform.

inform_version(*msg*)
Handle katcp v4 and below version inform.

inform_version_connect(*msg*)
Process a #version-connect message.

is_connected()
Check if the socket is currently connected.

Returns **connected** : bool

Whether the client is connected.

join(*timeout=None*)
Rejoin the client thread.

Parameters **timeout** : float in seconds

Seconds to wait for thread to finish.

Notes

Does nothing if the ioloop is not managed. Use until_stopped() instead.

notify_connected(*connected*)
Event handler that is called whenever the connection status changes.

Override in derived class for desired behaviour.

Note: This function should never block. Doing so will cause the client to cease processing data from the server until notify_connected completes.

Parameters **connected** : bool

Whether the client has just connected (True) or just disconnected (False).

preset_protocol_flags(*protocol_flags*)
Preset server protocol flags.

Sets the assumed server protocol flags and disables automatic server version detection.

Parameters **protocol_flags** : katcp.core.ProtocolFlags instance

request(*msg*, *use_mid=None*)
Send a request message, with automatic message ID assignment.

Parameters **msg** : katcp.Message request message

use_mid : bool or None, default=None

Returns **mid** : string or None

The message id, or None if no msg id is used

If use_mid is None and the server supports msg ids, or if use_mid is :

True a message ID will automatically be assigned msg.mid is None. :

if msg.mid has a value, and the server supports msg ids, that value :

will be used. If the server does not support msg ids, KatcpVersionError :

will be raised. :

running()
Whether the client is running.

Returns **running** : bool

Whether the client is running.

send_message(*msg*)
Send any kind of message.

Parameters **msg** : Message object

The message to send.

send_request(*msg*)
Send a request message.

Parameters **msg** : Message object

The request Message to send.

setDaemon(*daemonic*)
Set daemonic state of the managed ioloop thread to True / False

Calling this method for a non-managed ioloop has no effect. Must be called before start(), or it will also have no effect

set_ioloop(*ioloop=None*)
Set the tornado.ioloop.IOLoop instance to use.

This defaults to IOLoop.current(). If set_ioloop() is never called the IOLoop is managed: started in a new thread, and will be stopped if self.stop() is called.

Notes

Must be called before start() is called

start(*timeout=None*)
Start the client in a new thread.

Parameters **timeout** : float in seconds

Seconds to wait for client thread to start. Do not specify a timeout if start() is being called from the same ioloop that this client will be installed on, since it will block the ioloop without progressing.

stop(**args*, ***kwargs*)
Stop a running client.

If using a managed ioloop, this must be called from a different thread to the ioloop's. This method only returns once the client's main coroutine, *_install()*, has completed.

If using an unmanaged ioloop, this can be called from the same thread as the ioloop. The *until_stopped()* method can be used to wait on completion of the main coroutine, *_install()*.

Parameters **timeout** : float in seconds

Seconds to wait for both client thread to have *started*, and for stopping.

unhandled_inform(*msg*)
Fallback method for inform messages without a registered handler.

Parameters **msg** : Message object

The inform message that wasn't processed by any handlers.

unhandled_reply(*msg*)
Fallback method for reply messages without a registered handler.

Parameters **msg** : Message object

The reply message that wasn't processed by any handlers.

unhandled_request(*msg*)
Fallback method for requests without a registered handler.

Parameters **msg** : Message object

The request message that wasn't processed by any handlers.

until_connected(***kwargs*)
Return future that resolves when the client is connected.

until_protocol(***kwargs*)
Return future that resolves after receipt of katcp protocol info.

If the returned future resolves, the server's protocol information is available in the ProtocolFlags instance self.protocol_flags.

until_running(*timeout=None*)
Return future that resolves when the client is running.

Notes

Must be called from the same ioloop as the client.

until_stopped(*timeout=None*)
Return future that resolves when the client has stopped.

Parameters **timeout** : float in seconds

Seconds to wait for the client to stop.

Notes

If already running, *stop()* must be called before this.

Must be called from the same ioloop as the client. If using a different thread, or a managed ioloop, this method should not be used. Use *join()* instead.

Also note that stopped != not running. Stopped means the main coroutine has ended, or was never started. When stopping, the running flag is cleared some time before stopped is set.

wait_connected(*timeout=None*)
Wait until the client is connected.

Parameters **timeout** : float in seconds

Seconds to wait for the client to connect.

Returns **connected** : bool

Whether the client is connected.

Notes

Do not call this from the ioloop, use until_connected().

wait_disconnected(*timeout=None*)
Wait until the client is disconnected.

Parameters **timeout** : float in seconds

Seconds to wait for the client to disconnect.

Returns **disconnected** : bool

Whether the client is disconnected.

Notes

Do not call this from the ioloop, use until_disconnected().

wait_protocol(*timeout=None*)
Wait until katcp protocol information has been received from server.

Parameters **timeout** : float in seconds

Seconds to wait for the client to connect.

Returns **received** : bool

Whether protocol information was received

If this method returns True, the server's protocol information is :

available in the ProtocolFlags instance self.protocol_flags. :

Notes

Do not call this from the ioloop, use until_protocol().

wait_running(*timeout=None*)
Wait until the client is running.

Parameters **timeout** : float in seconds

Seconds to wait for the client to start running.

Returns **running** : bool

Whether the client is running

Notes

Do not call this from the ioloop, use until_running().

CallbackClient

class katcp.**CallbackClient**(*host*, *port*, *tb_limit=20*, *timeout=5.0*, *logger=<logging.Logger object>*, *auto_reconnect=True*)

Methods

CallbackClient.blocking_request(msg[, ...])	Send a request message and wait for its reply.
CallbackClient.callback_request(msg[, ...])	Send a request message.
CallbackClient.convert_seconds(time_seconds)	Convert a time in seconds to the device timestamp units.
CallbackClient.disconnect()	Force client connection to close, reconnect if auto-connect set.
CallbackClient.enable_thread_safety()	Enable thread-safety features.
CallbackClient.future_request(msg[, ...])	Send a request message, with future replies.
CallbackClient.handle_inform(msg)	Handle inform messages related to any current requests.
CallbackClient.handle_message(msg)	Handle a message from the server.
CallbackClient.handle_reply(msg)	Handle a reply message related to the current request.
CallbackClient.handle_request(msg)	Dispatch a request message to the appropriate method.
CallbackClient.inform_build_state(msg)	Handle katcp v4 and below build-state inform.
CallbackClient.inform_version(msg)	Handle katcp v4 and below version inform.

Continued on next page

Table 2 – continued from previous page

CallbackClient.inform_version_connect(msg)	Process a #version-connect message.
CallbackClient.is_connected()	Check if the socket is currently connected.
CallbackClient.join([timeout])	Rejoin the client thread.
CallbackClient.next()	
CallbackClient.notify_connected(connected)	Event handler that is called whenever the connection status changes.
CallbackClient.preset_protocol_flags(...)	Preset server protocol flags.
CallbackClient.request(msg[, use_mid])	Send a request message, with automatic message ID assignment.
CallbackClient.running()	Whether the client is running.
CallbackClient.send_message(msg)	Send any kind of message.
CallbackClient.send_request(msg)	Send a request message.
CallbackClient.setDaemon(daemonic)	Set daemonic state of the managed ioloop thread to True / False
CallbackClient.set_ioloop([ioloop])	Set the tornado.ioloop.IOLoop instance to use.
CallbackClient.start([timeout])	Start the client in a new thread.
CallbackClient.stop(*args, **kwargs)	Stop a running client.
CallbackClient.unhandled_inform(msg)	Fallback method for inform messages without a registered handler.
CallbackClient.unhandled_reply(msg)	Fallback method for reply messages without a registered handler.
CallbackClient.unhandled_request(msg)	Fallback method for requests without a registered handler.
CallbackClient.until_connected(**kwargs)	Return future that resolves when the client is connected.
CallbackClient.until_protocol(**kwargs)	Return future that resolves after receipt of katcp protocol info.
CallbackClient.until_running([timeout])	Return future that resolves when the client is running.
CallbackClient.until_stopped([timeout])	Return future that resolves when the client has stopped.
CallbackClient.wait_connected([timeout])	Wait until the client is connected.
CallbackClient.wait_disconnected([timeout])	Wait until the client is disconnected.
CallbackClient.wait_protocol([timeout])	Wait until katcp protocol information has been received from server.
CallbackClient.wait_running([timeout])	Wait until the client is running.

bind_address
(host, port) where the client is connecting

blocking_request(*msg*, *timeout=None*, *use_mid=None*)
Send a request message and wait for its reply.

Parameters **msg** : Message object

The request Message to send.

timeout : float in seconds

How long to wait for a reply. The default is the the timeout set when creating the AsyncClient.

use_mid : boolean, optional

Whether to use message IDs. Default is to use message IDs if the server supports them.

Returns **reply** : Message object

The reply message received.

informs : list of Message objects

A list of the inform messages received.

callback_request(*msg*, *reply_cb=None*, *inform_cb=None*, *user_data=None*, *timeout=None*, *use_mid=None*)

Send a request message.

Parameters **msg** : Message object

The request message to send.

reply_cb : function

The reply callback with signature reply_cb(msg) or reply_cb(msg, *user_data)

inform_cb : function

The inform callback with signature inform_cb(msg) or inform_cb(msg, *user_data)

user_data : tuple

Optional user data to send to the reply and inform callbacks.

timeout : float in seconds

How long to wait for a reply. The default is the the timeout set when creating the AsyncClient.

use_mid : boolean, optional

Whether to use message IDs. Default is to use message IDs if the server supports them.

convert_seconds(*time_seconds*)

Convert a time in seconds to the device timestamp units.

KATCP v4 and earlier, specified all timestamps in milliseconds. Since KATCP v5, all timestamps are in seconds. If the device KATCP version has been detected, this method converts a value in seconds to the appropriate (seconds or milliseconds) quantity. For version smaller than V4, the time value will be truncated to the nearest millisecond.

disconnect()

Force client connection to close, reconnect if auto-connect set.

enable_thread_safety()

Enable thread-safety features.

Must be called before start().

future_request(*msg*, *timeout=None*, *use_mid=None*)

Send a request message, with future replies.

Parameters **msg** : Message object

The request Message to send.

timeout : float in seconds

How long to wait for a reply. The default is the the timeout set when creating the AsyncClient.

use_mid : boolean, optional

Whether to use message IDs. Default is to use message IDs if the server supports them.

Returns A tornado.concurrent.Future that resolves with: :

reply : Message object

The reply message received.

informs : list of Message objects

A list of the inform messages received.

handle_inform(*msg*)
Handle inform messages related to any current requests.

Inform messages not related to the current request go up to the base class method.

Parameters msg : Message object

The inform message to dispatch.

handle_message(*msg*)
Handle a message from the server.

Parameters msg : Message object

The Message to dispatch to the handler methods.

handle_reply(*msg*)
Handle a reply message related to the current request.

Reply messages not related to the current request go up to the base class method.

Parameters msg : Message object

The reply message to dispatch.

handle_request(*msg*)
Dispatch a request message to the appropriate method.

Parameters msg : Message object

The request message to dispatch.

inform_build_state(*msg*)
Handle katcp v4 and below build-state inform.

inform_version(*msg*)
Handle katcp v4 and below version inform.

inform_version_connect(*msg*)
Process a #version-connect message.

is_connected()
Check if the socket is currently connected.

Returns connected : bool

Whether the client is connected.

join(*timeout=None*)
Rejoin the client thread.

Parameters timeout : float in seconds

Seconds to wait for thread to finish.

Notes

Does nothing if the ioloop is not managed. Use until_stopped() instead.

notify_connected(*connected*)
Event handler that is called whenever the connection status changes.

Override in derived class for desired behaviour.

Note: This function should never block. Doing so will cause the client to cease processing data from the server until notify_connected completes.

Parameters **connected** : bool

Whether the client has just connected (True) or just disconnected (False).

preset_protocol_flags(*protocol_flags*)
Preset server protocol flags.

Sets the assumed server protocol flags and disables automatic server version detection.

Parameters **protocol_flags** : katcp.core.ProtocolFlags instance

request(*msg*, *use_mid=None*)
Send a request message, with automatic message ID assignment.

Parameters **msg** : katcp.Message request message

use_mid : bool or None, default=None

Returns **mid** : string or None

The message id, or None if no msg id is used

If use_mid is None and the server supports msg ids, or if use_mid is :

True a message ID will automatically be assigned msg.mid is None. :

if msg.mid has a value, and the server supports msg ids, that value :

will be used. If the server does not support msg ids, KatcpVersionError :

will be raised. :

running()
Whether the client is running.

Returns **running** : bool

Whether the client is running.

send_message(*msg*)
Send any kind of message.

Parameters **msg** : Message object

The message to send.

send_request(*msg*)
Send a request message.

Parameters **msg** : Message object

The request Message to send.

setDaemon(*daemonic*)
Set daemonic state of the managed ioloop thread to True / False

Calling this method for a non-managed ioloop has no effect. Must be called before start(), or it will also have no effect

set_ioloop(*ioloop=None*)
Set the tornado.ioloop.IOLoop instance to use.

This defaults to IOLoop.current(). If set_ioloop() is never called the IOLoop is managed: started in a new thread, and will be stopped if self.stop() is called.

Notes

Must be called before start() is called

start(*timeout=None*)
Start the client in a new thread.

Parameters **timeout** : float in seconds

Seconds to wait for client thread to start. Do not specify a timeout if start() is being called from the same ioloop that this client will be installed on, since it will block the ioloop without progressing.

stop(**args*, ***kwargs*)
Stop a running client.

If using a managed ioloop, this must be called from a different thread to the ioloop's. This method only returns once the client's main coroutine, *_install()*, has completed.

If using an unmanaged ioloop, this can be called from the same thread as the ioloop. The *until_stopped()* method can be used to wait on completion of the main coroutine, *_install()*.

Parameters **timeout** : float in seconds

Seconds to wait for both client thread to have *started*, and for stopping.

unhandled_inform(*msg*)
Fallback method for inform messages without a registered handler.

Parameters **msg** : Message object

The inform message that wasn't processed by any handlers.

unhandled_reply(*msg*)
Fallback method for reply messages without a registered handler.

Parameters **msg** : Message object

The reply message that wasn't processed by any handlers.

unhandled_request(*msg*)
Fallback method for requests without a registered handler.

Parameters **msg** : Message object

The request message that wasn't processed by any handlers.

until_connected(***kwargs*)
Return future that resolves when the client is connected.

until_protocol(***kwargs*)
Return future that resolves after receipt of katcp protocol info.

If the returned future resolves, the server's protocol information is available in the ProtocolFlags instance self.protocol_flags.

until_running(*timeout=None*)
Return future that resolves when the client is running.

Notes

Must be called from the same ioloop as the client.

until_stopped(*timeout=None*)
Return future that resolves when the client has stopped.

Parameters **timeout** : float in seconds

Seconds to wait for the client to stop.

Notes

If already running, *stop()* must be called before this.

Must be called from the same ioloop as the client. If using a different thread, or a managed ioloop, this method should not be used. Use *join()* instead.

Also note that stopped != not running. Stopped means the main coroutine has ended, or was never started. When stopping, the running flag is cleared some time before stopped is set.

wait_connected(*timeout=None*)
Wait until the client is connected.

Parameters **timeout** : float in seconds

Seconds to wait for the client to connect.

Returns **connected** : bool

Whether the client is connected.

Notes

Do not call this from the ioloop, use until_connected().

wait_disconnected(*timeout=None*)
Wait until the client is disconnected.

Parameters **timeout** : float in seconds

Seconds to wait for the client to disconnect.

Returns **disconnected** : bool

Whether the client is disconnected.

Notes

Do not call this from the ioloop, use until_disconnected().

wait_protocol(*timeout=None*)
Wait until katcp protocol information has been received from server.

Parameters **timeout** : float in seconds

Seconds to wait for the client to connect.

Returns **received** : bool

Whether protocol information was received

If this method returns True, the server's protocol information is :

available in the ProtocolFlags instance self.protocol_flags. :

Notes

Do not call this from the ioloop, use until_protocol().

wait_running(*timeout=None*)
Wait until the client is running.

Parameters **timeout** : float in seconds

Seconds to wait for the client to start running.

Returns **running** : bool

Whether the client is running

Notes

Do not call this from the ioloop, use until_running().

AsyncClient

class katcp.**AsyncClient**(*host*, *port*, *tb_limit=20*, *timeout=5.0*, *logger=<logging.Logger object>*, *auto_reconnect=True*)
Implement async and callback-based requests on top of DeviceClient.

This client will use message IDs if the server supports them.

Parameters **host** : string

Host to connect to.

port : int

Port to connect to.

tb_limit : int, optional

Maximum number of stack frames to send in error traceback.

logger : object, optional

Python Logger object to log to. Default is a logger named 'katcp'.

auto_reconnect : bool, optional

Whether to automatically reconnect if the connection dies.

timeout : float in seconds, optional

Default number of seconds to wait before a callback callback_request times out. Can be overridden in individual calls to callback_request.

Examples

```
>>> def reply_cb(msg):
...     print "Reply:", msg
...
>>> def inform_cb(msg):
...     print "Inform:", msg
...
>>> c = AsyncClient('localhost', 10000)
>>> c.start()
>>> c.ioloop.add_callback(
...     c.callback_request,
...     katcp.Message.request('myreq'),
...     reply_cb=reply_cb,
...     inform_cb=inform_cb,
... )
...
>>> # expect reply to be printed here
>>> # stop the client once we're finished with it
>>> c.stop()
>>> c.join()
```

Methods

AsyncClient.blocking_request(msg[, timeout, ...])	Send a request message and wait for its reply.
AsyncClient.callback_request(msg[, ...])	Send a request message.
AsyncClient.convert_seconds(time_seconds)	Convert a time in seconds to the device timestamp units.
AsyncClient.disconnect()	Force client connection to close, reconnect if auto-connect set.
AsyncClient.enable_thread_safety()	Enable thread-safety features.
AsyncClient.future_request(msg[, timeout, ...])	Send a request message, with future replies.
AsyncClient.handle_inform(msg)	Handle inform messages related to any current requests.
AsyncClient.handle_message(msg)	Handle a message from the server.
AsyncClient.handle_reply(msg)	Handle a reply message related to the current request.
AsyncClient.handle_request(msg)	Dispatch a request message to the appropriate method.
AsyncClient.inform_build_state(msg)	Handle katcp v4 and below build-state inform.
AsyncClient.inform_version(msg)	Handle katcp v4 and below version inform.
AsyncClient.inform_version_connect(msg)	Process a #version-connect message.

Continued on next page

Table 3 – continued from previous page

AsyncClient.is_connected()	Check if the socket is currently connected.
AsyncClient.join([timeout])	Rejoin the client thread.
AsyncClient.next()	
AsyncClient.notify_connected(connected)	Event handler that is called whenever the connection status changes.
AsyncClient.preset_protocol_flags(protocol_flags)	Pre-set the server protocol flags.
AsyncClient.request(msg[, use_mid])	Send a request message, with automatic message ID assignment.
AsyncClient.running()	Whether the client is running.
AsyncClient.send_message(msg)	Send any kind of message.
AsyncClient.send_request(msg)	Send a request message.
AsyncClient.set_ioloop([ioloop])	Set the tornado.ioloop.IOLoop instance to use.
AsyncClient.start([timeout])	Start the client in a new thread.
AsyncClient.stop(*args, **kwargs)	Stop a running client.
AsyncClient.unhandled_inform(msg)	Fallback method for inform messages without a registered handler.
AsyncClient.unhandled_reply(msg)	Fallback method for reply messages without a registered handler.
AsyncClient.unhandled_request(msg)	Fallback method for requests without a registered handler.
AsyncClient.until_connected(**kwargs)	Return future that resolves when the client is connected.
AsyncClient.until_protocol(**kwargs)	Return future that resolves after receipt of katcp protocol info.
AsyncClient.until_running([timeout])	Return future that resolves when the client is running.
AsyncClient.until_stopped([timeout])	Return future that resolves when the client has stopped.
AsyncClient.wait_connected([timeout])	Wait until the client is connected.
AsyncClient.wait_disconnected([timeout])	Wait until the client is disconnected.
AsyncClient.wait_protocol([timeout])	Wait until katcp protocol information has been received from server.
AsyncClient.wait_running([timeout])	Wait until the client is running.

bind_address
(host, port) where the client is connecting

blocking_request(*msg*, *timeout=None*, *use_mid=None*)
Send a request message and wait for its reply.

Parameters **msg** : Message object

The request Message to send.

timeout : float in seconds

How long to wait for a reply. The default is the the timeout set when creating the AsyncClient.

use_mid : boolean, optional

Whether to use message IDs. Default is to use message IDs if the server supports them.

Returns **reply** : Message object

The reply message received.

informs : list of Message objects

A list of the inform messages received.

callback_request(*msg*, *reply_cb=None*, *inform_cb=None*, *user_data=None*, *timeout=None*, *use_mid=None*)

Send a request message.

Parameters **msg** : Message object

The request message to send.

reply_cb : function

The reply callback with signature reply_cb(msg) or reply_cb(msg, *user_data)

inform_cb : function

The inform callback with signature inform_cb(msg) or inform_cb(msg, *user_data)

user_data : tuple

Optional user data to send to the reply and inform callbacks.

timeout : float in seconds

How long to wait for a reply. The default is the the timeout set when creating the AsyncClient.

use_mid : boolean, optional

Whether to use message IDs. Default is to use message IDs if the server supports them.

convert_seconds(*time_seconds*)

Convert a time in seconds to the device timestamp units.

KATCP v4 and earlier, specified all timestamps in milliseconds. Since KATCP v5, all timestamps are in seconds. If the device KATCP version has been detected, this method converts a value in seconds to the appropriate (seconds or milliseconds) quantity. For version smaller than V4, the time value will be truncated to the nearest millisecond.

disconnect()

Force client connection to close, reconnect if auto-connect set.

enable_thread_safety()

Enable thread-safety features.

Must be called before start().

future_request(*msg*, *timeout=None*, *use_mid=None*)

Send a request message, with future replies.

Parameters **msg** : Message object

The request Message to send.

timeout : float in seconds

How long to wait for a reply. The default is the the timeout set when creating the AsyncClient.

use_mid : boolean, optional

Whether to use message IDs. Default is to use message IDs if the server supports them.

Returns **A tornado.concurrent.Future that resolves with:** :

reply : Message object

The reply message received.

informs : list of Message objects

A list of the inform messages received.

handle_inform(*msg*)
Handle inform messages related to any current requests.

Inform messages not related to the current request go up to the base class method.

Parameters **msg** : Message object

The inform message to dispatch.

handle_message(*msg*)
Handle a message from the server.

Parameters **msg** : Message object

The Message to dispatch to the handler methods.

handle_reply(*msg*)
Handle a reply message related to the current request.

Reply messages not related to the current request go up to the base class method.

Parameters **msg** : Message object

The reply message to dispatch.

handle_request(*msg*)
Dispatch a request message to the appropriate method.

Parameters **msg** : Message object

The request message to dispatch.

inform_build_state(*msg*)
Handle katcp v4 and below build-state inform.

inform_version(*msg*)
Handle katcp v4 and below version inform.

inform_version_connect(*msg*)
Process a #version-connect message.

is_connected()
Check if the socket is currently connected.

Returns **connected** : bool

Whether the client is connected.

join(*timeout=None*)
Rejoin the client thread.

Parameters **timeout** : float in seconds

Seconds to wait for thread to finish.

Notes

Does nothing if the ioloop is not managed. Use until_stopped() instead.

notify_connected(*connected*)
Event handler that is called whenever the connection status changes.

Override in derived class for desired behaviour.

Note: This function should never block. Doing so will cause the client to cease processing data from the server until notify_connected completes.

Parameters **connected** : bool

Whether the client has just connected (True) or just disconnected (False).

preset_protocol_flags(*protocol_flags*)
Preset server protocol flags.

Sets the assumed server protocol flags and disables automatic server version detection.

Parameters **protocol_flags** : katcp.core.ProtocolFlags instance

request(*msg*, *use_mid=None*)
Send a request message, with automatic message ID assignment.

Parameters **msg** : katcp.Message request message

use_mid : bool or None, default=None

Returns **mid** : string or None

The message id, or None if no msg id is used

If use_mid is None and the server supports msg ids, or if use_mid is :

True a message ID will automatically be assigned msg.mid is None. :

if msg.mid has a value, and the server supports msg ids, that value :

will be used. If the server does not support msg ids, KatcpVersionError :

will be raised. :

running()
Whether the client is running.

Returns **running** : bool

Whether the client is running.

send_message(*msg*)
Send any kind of message.

Parameters **msg** : Message object

The message to send.

send_request(*msg*)
Send a request message.

Parameters **msg** : Message object

The request Message to send.

set_ioloop(*ioloop=None*)
Set the tornado.ioloop.IOLoop instance to use.

This defaults to IOLoop.current(). If set_ioloop() is never called the IOLoop is managed: started in a new thread, and will be stopped if self.stop() is called.

Notes

Must be called before start() is called

start(*timeout=None*)
Start the client in a new thread.

Parameters **timeout** : float in seconds

Seconds to wait for client thread to start. Do not specify a timeout if start() is being called from the same ioloop that this client will be installed on, since it will block the ioloop without progressing.

stop(**args*, ***kwargs*)
Stop a running client.

If using a managed ioloop, this must be called from a different thread to the ioloop's. This method only returns once the client's main coroutine, *_install()*, has completed.

If using an unmanaged ioloop, this can be called from the same thread as the ioloop. The *until_stopped()* method can be used to wait on completion of the main coroutine, *_install()*.

Parameters **timeout** : float in seconds

Seconds to wait for both client thread to have *started*, and for stopping.

unhandled_inform(*msg*)
Fallback method for inform messages without a registered handler.

Parameters **msg** : Message object

The inform message that wasn't processed by any handlers.

unhandled_reply(*msg*)
Fallback method for reply messages without a registered handler.

Parameters **msg** : Message object

The reply message that wasn't processed by any handlers.

unhandled_request(*msg*)
Fallback method for requests without a registered handler.

Parameters **msg** : Message object

The request message that wasn't processed by any handlers.

until_connected(***kwargs*)
Return future that resolves when the client is connected.

until_protocol(***kwargs*)
Return future that resolves after receipt of katcp protocol info.

If the returned future resolves, the server's protocol information is available in the ProtocolFlags instance self.protocol_flags.

until_running(*timeout=None*)
Return future that resolves when the client is running.

Notes

Must be called from the same ioloop as the client.

until_stopped(*timeout=None*)
Return future that resolves when the client has stopped.

Parameters **timeout** : float in seconds

Seconds to wait for the client to stop.

Notes

If already running, *stop()* must be called before this.

Must be called from the same ioloop as the client. If using a different thread, or a managed ioloop, this method should not be used. Use *join()* instead.

Also note that stopped != not running. Stopped means the main coroutine has ended, or was never started. When stopping, the running flag is cleared some time before stopped is set.

wait_connected(*timeout=None*)
Wait until the client is connected.

Parameters **timeout** : float in seconds

Seconds to wait for the client to connect.

Returns **connected** : bool

Whether the client is connected.

Notes

Do not call this from the ioloop, use until_connected().

wait_disconnected(*timeout=None*)
Wait until the client is disconnected.

Parameters **timeout** : float in seconds

Seconds to wait for the client to disconnect.

Returns **disconnected** : bool

Whether the client is disconnected.

Notes

Do not call this from the ioloop, use until_disconnected().

wait_protocol(*timeout=None*)
Wait until katcp protocol information has been received from server.

Parameters **timeout** : float in seconds

Seconds to wait for the client to connect.

Returns **received** : bool

Whether protocol information was received

If this method returns True, the server's protocol information is :

available in the ProtocolFlags instance self.protocol_flags. :

Notes

Do not call this from the ioloop, use until_protocol().

wait_running(*timeout=None*)

Wait until the client is running.

Parameters **timeout** : float in seconds

Seconds to wait for the client to start running.

Returns **running** : bool

Whether the client is running

Notes

Do not call this from the ioloop, use until_running().

Base Classes

class katcp.**DeviceClient**(*host*, *port*, *tb_limit=20*, *logger=<logging.Logger object>*, *auto_reconnect=True*)

Device client proxy.

Subclasses should implement .reply_*, .inform_* and send_request_* methods to take actions when messages arrive, and implement unhandled_inform, unhandled_reply and unhandled_request to provide fallbacks for messages for which there is no handler.

Request messages can be sent by calling .send_request().

Parameters **host** : string

Host to connect to.

port : int

Port to connect to.

tb_limit : int

Maximum number of stack frames to send in error traceback.

logger : object

Python Logger object to log to.

auto_reconnect : bool

Whether to automatically reconnect if the connection dies.

Notes

The client may block its ioloop if the default blocking tornado DNS resolver is used. When an ioloop is shared, it would make sense to configure one of the non-blocking resolver classes, see http://tornado.readthedocs.org/en/latest/netutil.html

Examples

```
>>> MyClient(DeviceClient):
...     def reply_myreq(self, msg):
...         print str(msg)
...
>>> c = MyClient('localhost', 10000){
>>> c.start()
>>> c.send_request(katcp.Message.request('myreq'))
>>> # expect reply to be printed here
>>> # stop the client once we're finished with it
>>> c.stop()
>>> c.join()
```

Methods

DeviceClient.convert_seconds(time_seconds)	Convert a time in seconds to the device timestamp units.
DeviceClient.disconnect()	Force client connection to close, reconnect if auto-connect set.
DeviceClient.enable_thread_safety()	Enable thread-safety features.
DeviceClient.handle_inform(msg)	Dispatch an inform message to the appropriate method.
DeviceClient.handle_message(msg)	Handle a message from the server.
DeviceClient.handle_reply(msg)	Dispatch a reply message to the appropriate method.
DeviceClient.handle_request(msg)	Dispatch a request message to the appropriate method.
DeviceClient.inform_build_state(msg)	Handle katcp v4 and below build-state inform.
DeviceClient.inform_version(msg)	Handle katcp v4 and below version inform.
DeviceClient.inform_version_connect(msg)	Process a #version-connect message.
DeviceClient.is_connected()	Check if the socket is currently connected.
DeviceClient.join([timeout])	Rejoin the client thread.
DeviceClient.next()	
DeviceClient.notify_connected(connected)	Event handler that is called whenever the connection status changes.
DeviceClient.preset_protocol_flags(...)	Preset server protocol flags.
DeviceClient.request(msg[, use_mid])	Send a request message, with automatic message ID assignment.
DeviceClient.running()	Whether the client is running.
DeviceClient.send_message(msg)	Send any kind of message.
DeviceClient.send_request(msg)	Send a request message.
DeviceClient.set_ioloop([ioloop])	Set the tornado.ioloop.IOLoop instance to use.
DeviceClient.start([timeout])	Start the client in a new thread.
DeviceClient.stop([timeout])	Stop a running client.
DeviceClient.unhandled_inform(msg)	Fallback method for inform messages without a registered handler.
DeviceClient.unhandled_reply(msg)	Fallback method for reply messages without a registered handler.
DeviceClient.unhandled_request(msg)	Fallback method for requests without a registered handler.

Continued on next page

Table 4 – continued from previous page

DeviceClient.until_connected(**kwargs)	Return future that resolves when the client is connected.
DeviceClient.until_protocol(**kwargs)	Return future that resolves after receipt of katcp protocol info.
DeviceClient.until_running([timeout])	Return future that resolves when the client is running.
DeviceClient.until_stopped([timeout])	Return future that resolves when the client has stopped.
DeviceClient.wait_connected([timeout])	Wait until the client is connected.
DeviceClient.wait_disconnected([timeout])	Wait until the client is disconnected.
DeviceClient.wait_protocol([timeout])	Wait until katcp protocol information has been received from server.
DeviceClient.wait_running([timeout])	Wait until the client is running.

MAX_LOOP_LATENCY = 0.03
Do not spend more than this many seconds reading pipelined socket data

IOStream inline-reading can result in ioloop starvation (see https://groups.google.com/forum/#!topic/python-tornado/yJrDAwDR_kA).

MAX_MSG_SIZE = 2097152
Maximum message size that can be received in bytes.

If more than MAX_MSG_SIZE bytes are read from the socket without encountering a message terminator (i.e. newline), the connection is closed.

MAX_WRITE_BUFFER_SIZE = 4194304
Maximum outstanding bytes to be buffered by the server process.

If more than MAX_WRITE_BUFFER_SIZE bytes are outstanding, the connection is closed. Note that the OS also buffers socket writes, so more than MAX_WRITE_BUFFER_SIZE bytes may be untransmitted in total.

bind_address
(host, port) where the client is connecting

convert_seconds(*time_seconds*)
Convert a time in seconds to the device timestamp units.

KATCP v4 and earlier, specified all timestamps in milliseconds. Since KATCP v5, all timestamps are in seconds. If the device KATCP version has been detected, this method converts a value in seconds to the appropriate (seconds or milliseconds) quantity. For version smaller than V4, the time value will be truncated to the nearest millisecond.

disconnect()
Force client connection to close, reconnect if auto-connect set.

enable_thread_safety()
Enable thread-safety features.

Must be called before start().

handle_inform(*msg*)
Dispatch an inform message to the appropriate method.

Parameters **msg** : Message object

The inform message to dispatch.

handle_message(*msg*)
Handle a message from the server.

Parameters **msg** : Message object

The Message to dispatch to the handler methods.

handle_reply(*msg*)
Dispatch a reply message to the appropriate method.

Parameters **msg** : Message object

The reply message to dispatch.

handle_request(*msg*)
Dispatch a request message to the appropriate method.

Parameters **msg** : Message object

The request message to dispatch.

inform_build_state(*msg*)
Handle katcp v4 and below build-state inform.

inform_version(*msg*)
Handle katcp v4 and below version inform.

inform_version_connect(*msg*)
Process a #version-connect message.

is_connected()
Check if the socket is currently connected.

Returns **connected** : bool

Whether the client is connected.

join(*timeout=None*)
Rejoin the client thread.

Parameters **timeout** : float in seconds

Seconds to wait for thread to finish.

Notes

Does nothing if the ioloop is not managed. Use until_stopped() instead.

notify_connected(*connected*)
Event handler that is called whenever the connection status changes.

Override in derived class for desired behaviour.

Note: This function should never block. Doing so will cause the client to cease processing data from the server until notify_connected completes.

Parameters **connected** : bool

Whether the client has just connected (True) or just disconnected (False).

preset_protocol_flags(*protocol_flags*)
Preset server protocol flags.

Sets the assumed server protocol flags and disables automatic server version detection.

Parameters **protocol_flags** : katcp.core.ProtocolFlags instance

request(*msg*, *use_mid=None*)
Send a request message, with automatic message ID assignment.

Parameters **msg** : katcp.Message request message

use_mid : bool or None, default=None

Returns **mid** : string or None

The message id, or None if no msg id is used

If use_mid is None and the server supports msg ids, or if use_mid is :

True a message ID will automatically be assigned msg.mid is None. :

if msg.mid has a value, and the server supports msg ids, that value :

will be used. If the server does not support msg ids, KatcpVersionError :

will be raised. :

running()
Whether the client is running.

Returns **running** : bool

Whether the client is running.

send_message(*msg*)
Send any kind of message.

Parameters **msg** : Message object

The message to send.

send_request(*msg*)
Send a request message.

Parameters **msg** : Message object

The request Message to send.

set_ioloop(*ioloop=None*)
Set the tornado.ioloop.IOLoop instance to use.

This defaults to IOLoop.current(). If set_ioloop() is never called the IOLoop is managed: started in a new thread, and will be stopped if self.stop() is called.

Notes

Must be called before start() is called

start(*timeout=None*)
Start the client in a new thread.

Parameters **timeout** : float in seconds

Seconds to wait for client thread to start. Do not specify a timeout if start() is being called from the same ioloop that this client will be installed on, since it will block the ioloop without progressing.

stop(*timeout=None*)
Stop a running client.

If using a managed ioloop, this must be called from a different thread to the ioloop's. This method only returns once the client's main coroutine, *_install()*, has completed.

If using an unmanaged ioloop, this can be called from the same thread as the ioloop. The *until_stopped()* method can be used to wait on completion of the main coroutine, *_install()*.

Parameters **timeout** : float in seconds

Seconds to wait for both client thread to have *started*, and for stopping.

unhandled_inform(*msg*)
Fallback method for inform messages without a registered handler.

Parameters **msg** : Message object

The inform message that wasn't processed by any handlers.

unhandled_reply(*msg*)
Fallback method for reply messages without a registered handler.

Parameters **msg** : Message object

The reply message that wasn't processed by any handlers.

unhandled_request(*msg*)
Fallback method for requests without a registered handler.

Parameters **msg** : Message object

The request message that wasn't processed by any handlers.

until_connected(***kwargs*)
Return future that resolves when the client is connected.

until_protocol(***kwargs*)
Return future that resolves after receipt of katcp protocol info.

If the returned future resolves, the server's protocol information is available in the ProtocolFlags instance self.protocol_flags.

until_running(*timeout=None*)
Return future that resolves when the client is running.

Notes

Must be called from the same ioloop as the client.

until_stopped(*timeout=None*)
Return future that resolves when the client has stopped.

Parameters **timeout** : float in seconds

Seconds to wait for the client to stop.

Notes

If already running, *stop()* must be called before this.

Must be called from the same ioloop as the client. If using a different thread, or a managed ioloop, this method should not be used. Use *join()* instead.

Also note that stopped != not running. Stopped means the main coroutine has ended, or was never started. When stopping, the running flag is cleared some time before stopped is set.

wait_connected(*timeout=None*)
Wait until the client is connected.

Parameters **timeout** : float in seconds

Seconds to wait for the client to connect.

Returns **connected** : bool

Whether the client is connected.

Notes

Do not call this from the ioloop, use until_connected().

wait_disconnected(*timeout=None*)
Wait until the client is disconnected.

Parameters **timeout** : float in seconds

Seconds to wait for the client to disconnect.

Returns **disconnected** : bool

Whether the client is disconnected.

Notes

Do not call this from the ioloop, use until_disconnected().

wait_protocol(*timeout=None*)
Wait until katcp protocol information has been received from server.

Parameters **timeout** : float in seconds

Seconds to wait for the client to connect.

Returns **received** : bool

Whether protocol information was received

If this method returns True, the server's protocol information is :

available in the ProtocolFlags instance self.protocol_flags. :

Notes

Do not call this from the ioloop, use until_protocol().

wait_running(*timeout=None*)
Wait until the client is running.

Parameters **timeout** : float in seconds

Seconds to wait for the client to start running.

Returns **running** : bool

Whether the client is running

Notes

Do not call this from the ioloop, use until_running().

Exceptions

class katcp.**KatcpClientError**
Raised by KATCP clients when an error occurs.

1.2.2 Server

AsyncDeviceServer

class katcp.**AsyncDeviceServer**(**args*, ***kwargs*)
DeviceServer that is automatically configured for async use.

Same as instantiating a *DeviceServer* instance and calling methods *set_concurrency_options(thread_safe=False, handler_thread=False)* and *set_ioloop(tornado.ioloop.IOLoop.current())* before starting.

Methods

AsyncDeviceServer.add_sensor(sensor)	Add a sensor to the device.
AsyncDeviceServer.build_state()	Return build state string of the form name-major.minor[(a|b|rc)n].
AsyncDeviceServer.clear_strategies(client_conn)	Clear the sensor strategies of a client connection.
AsyncDeviceServer.create_exception_reply_and_log(...)	
AsyncDeviceServer.create_log_inform(...[, ...])	Create a katcp logging inform message.
AsyncDeviceServer.get_sensor(sensor_name)	Fetch the sensor with the given name.
AsyncDeviceServer.get_sensors()	Fetch a list of all sensors.
AsyncDeviceServer.handle_inform(connection, msg)	Dispatch an inform message to the appropriate method.
AsyncDeviceServer.handle_message(...)	Handle messages of all types from clients.
AsyncDeviceServer.handle_reply(connection, msg)	Dispatch a reply message to the appropriate method.
AsyncDeviceServer.handle_request(connection, msg)	Dispatch a request message to the appropriate method.

Continued on next page

Table 5 – continued from previous page

AsyncDeviceServer.has_sensor(sensor_name)	Whether the sensor with specified name is known.
AsyncDeviceServer.inform(connection, msg)	Send an inform message to a particular client.
AsyncDeviceServer.join([timeout])	Rejoin the server thread.
AsyncDeviceServer.mass_inform(msg)	Send an inform message to all clients.
AsyncDeviceServer.next()	
AsyncDeviceServer.on_client_connect(**kwargs)	Inform client of build state and version on connect.
AsyncDeviceServer.on_client_disconnect(...)	Inform client it is about to be disconnected.
AsyncDeviceServer.on_message(client_conn, msg)	Dummy implementation of on_message required by KATCPServer.
AsyncDeviceServer.remove_sensor(sensor)	Remove a sensor from the device.
AsyncDeviceServer.reply(connection, reply, ...)	Send an asynchronous reply to an earlier request.
AsyncDeviceServer.reply_inform(connection, ...)	Send an inform as part of the reply to an earlier request.
AsyncDeviceServer.request_client_list(req, msg)	Request the list of connected clients.
AsyncDeviceServer.request_halt(req, msg)	Halt the device server.
AsyncDeviceServer.request_help(req, msg)	Return help on the available requests.
AsyncDeviceServer.request_log_level(req, msg)	Query or set the current logging level.
AsyncDeviceServer.request_request_timeout_hint(...)	Return timeout hints for requests
AsyncDeviceServer.request_restart(req, msg)	Restart the device server.
AsyncDeviceServer.request_sensor_list(req, msg)	Request the list of sensors.
AsyncDeviceServer.request_sensor_sampling(...)	Configure or query the way a sensor is sampled.
AsyncDeviceServer.request_sensor_sampling_clear(...)	Set all sampling strategies for this client to none.
AsyncDeviceServer.request_sensor_value(req, msg)	Request the value of a sensor or sensors.
AsyncDeviceServer.request_version_list(req, msg)	Request the list of versions of roles and subcomponents.
AsyncDeviceServer.request_watchdog(req, msg)	Check that the server is still alive.
AsyncDeviceServer.running()	Whether the server is running.
AsyncDeviceServer.setDaemon(daemonic)	Set daemonic state of the managed ioloop thread to True / False
AsyncDeviceServer.set_concurrency_options([...])	Set concurrency options for this device server.
AsyncDeviceServer.set_ioloop([ioloop])	Set the tornado IOLoop to use.
AsyncDeviceServer.set_restart_queue(...)	Set the restart queue.

Continued on next page

Table 5 – continued from previous page

AsyncDeviceServer.setup_sensors()	Populate the dictionary of sensors.
AsyncDeviceServer.start([timeout])	Start the server in a new thread.
AsyncDeviceServer.stop([timeout])	Stop a running server (from another thread).
AsyncDeviceServer.sync_with_ioloop([timeout])	Block for ioloop to complete a loop if called from another thread.
AsyncDeviceServer.version()	Return a version string of the form type-major.minor.
AsyncDeviceServer.wait_running([timeout])	Wait until the server is running

add_sensor(*sensor*)
Add a sensor to the device.

Usually called inside .setup_sensors() but may be called from elsewhere.

Parameters **sensor** : Sensor object

The sensor object to register with the device server.

build_state()
Return build state string of the form name-major.minor[(a|b|rc)n].

clear_strategies(*client_conn*, *remove_client=False*)
Clear the sensor strategies of a client connection.

Parameters **client_connection** : ClientConnection instance

The connection that should have its sampling strategies cleared

remove_client : bool, optional

Remove the client connection from the strategies data-structure. Useful for clients that disconnect.

create_log_inform(*level_name*, *msg*, *name*, *timestamp=None*)
Create a katcp logging inform message.

Usually this will be called from inside a DeviceLogger object, but it is also used by the methods in this class when errors need to be reported to the client.

get_sensor(*sensor_name*)
Fetch the sensor with the given name.

Parameters **sensor_name** : str

Name of the sensor to retrieve.

Returns **sensor** : Sensor object

The sensor with the given name.

get_sensors()
Fetch a list of all sensors.

Returns **sensors** : list of Sensor objects

The list of sensors registered with the device server.

handle_inform(*connection*, *msg*)
Dispatch an inform message to the appropriate method.

Parameters **connection** : ClientConnection object

The client connection the message was from.

msg : Message object

The inform message to process.

handle_message(*client_conn*, *msg*)
Handle messages of all types from clients.

Parameters **client_conn** : ClientConnection object

The client connection the message was from.

msg : Message object

The message to process.

handle_reply(*connection*, *msg*)
Dispatch a reply message to the appropriate method.

Parameters **connection** : ClientConnection object

The client connection the message was from.

msg : Message object

The reply message to process.

handle_request(*connection*, *msg*)
Dispatch a request message to the appropriate method.

Parameters **connection** : ClientConnection object

The client connection the message was from.

msg : Message object

The request message to process.

Returns **done_future** : Future or None

Returns Future for async request handlers that will resolve when done, or None for sync request handlers once they have completed.

has_sensor(*sensor_name*)
Whether the sensor with specified name is known.

inform(*connection*, *msg*)
Send an inform message to a particular client.

Should only be used for asynchronous informs. Informs that are part of the response to a request should use *reply_inform()* so that the message identifier from the original request can be attached to the inform.

Parameters **connection** : ClientConnection object

The client to send the message to.

msg : Message object

The inform message to send.

join(*timeout=None*)
Rejoin the server thread.

Parameters **timeout** : float or None, optional

Time in seconds to wait for the thread to finish.

mass_inform(*msg*)
Send an inform message to all clients.

Parameters **msg** : Message object

The inform message to send.

on_client_connect(***kwargs*)
Inform client of build state and version on connect.

Parameters **client_conn** : ClientConnection object

The client connection that has been successfully established.

Returns **Future that resolves when the device is ready to accept messages.** :

on_client_disconnect(*client_conn*, *msg*, *connection_valid*)
Inform client it is about to be disconnected.

Parameters **client_conn** : ClientConnection object

The client connection being disconnected.

msg : str

Reason client is being disconnected.

connection_valid : bool

True if connection is still open for sending, False otherwise.

Returns **Future that resolves when the client connection can be closed.** :

on_message(*client_conn*, *msg*)
Dummy implementation of on_message required by KATCPServer.

Will be replaced by a handler with the appropriate concurrency semantics when set_concurrency_options is called (defaults are set in __init__()).

remove_sensor(*sensor*)
Remove a sensor from the device.

Also deregisters all clients observing the sensor.

Parameters **sensor** : Sensor object or name string

The sensor to remove from the device server.

reply(*connection*, *reply*, *orig_req*)
Send an asynchronous reply to an earlier request.

Parameters **connection** : ClientConnection object

The client to send the reply to.

reply : Message object

The reply message to send.

orig_req : Message object

The request message being replied to. The reply message's id is overridden with the id from orig_req before the reply is sent.

reply_inform(*connection*, *inform*, *orig_req*)
Send an inform as part of the reply to an earlier request.

Parameters **connection** : ClientConnection object

The client to send the inform to.

inform : Message object

The inform message to send.

orig_req : Message object

The request message being replied to. The inform message's id is overridden with the id from orig_req before the inform is sent.

request_client_list(*req*, *msg*)
Request the list of connected clients.

The list of clients is sent as a sequence of #client-list informs.

Informs **addr** : str

The address of the client as host:port with host in dotted quad notation. If the address of the client could not be determined (because, for example, the client disconnected suddenly) then a unique string representing the client is sent instead.

Returns **success** : {'ok', 'fail'}

Whether sending the client list succeeded.

informs : int

Number of #client-list inform messages sent.

Examples

```
?client-list
#client-list 127.0.0.1:53600
!client-list ok 1
```

request_halt(*req*, *msg*)
Halt the device server.

Returns **success** : {'ok', 'fail'}

Whether scheduling the halt succeeded.

Examples

```
?halt
!halt ok
```

request_help(*req*, *msg*)
Return help on the available requests.

Return a description of the available requests using a sequence of #help informs.

Parameters **request** : str, optional

The name of the request to return help for (the default is to return help for all requests).

Informs **request** : str

The name of a request.

description : str

Documentation for the named request.

Returns **success** : {'ok', 'fail'}

Whether sending the help succeeded.

informs : int

Number of #help inform messages sent.

Examples

```
?help
#help halt ...description...
#help help ...description...
...
!help ok 5

?help halt
#help halt ...description...
!help ok 1
```

request_log_level (*req*, *msg*)

Query or set the current logging level.

Parameters **level** : {'all', 'trace', 'debug', 'info', 'warn', 'error', 'fatal', 'off'}, optional

Name of the logging level to set the device server to (the default is to leave the log level unchanged).

Returns **success** : {'ok', 'fail'}

Whether the request succeeded.

level : {'all', 'trace', 'debug', 'info', 'warn', 'error', 'fatal', 'off'}

The log level after processing the request.

Examples

```
?log-level
!log-level ok warn

?log-level info
!log-level ok info
```

request_request_timeout_hint (*req*, *msg*)

Return timeout hints for requests

KATCP requests should generally take less than 5s to complete, but some requests are unavoidably slow. This results in spurious client timeout errors. This request provides timeout hints that clients can use to select suitable request timeouts.

Parameters **request** : str, optional

The name of the request to return a timeout hint for (the default is to return hints for all requests that have timeout hints). Returns one inform per request. Must be an existing request if specified.

Informs **request** : str

The name of the request.

suggested_timeout : float

Suggested request timeout in seconds for the request. If *suggested_timeout* is zero (0), no timeout hint is available.

Returns **success** : {'ok', 'fail'}

Whether sending the help succeeded.

informs : int

Number of #request-timeout-hint inform messages sent.

Notes

?request-timeout-hint without a parameter will only return informs for requests that have specific timeout hints, so it will most probably be a subset of all the requests, or even no informs at all.

Examples

```
?request-timeout-hint
#request-timeout-hint halt 5
#request-timeout-hint very-slow-request 500
...
!request-timeout-hint ok 5

?request-timeout-hint moderately-slow-request
#request-timeout-hint moderately-slow-request 20
!request-timeout-hint ok 1
```

request_restart(*req*, *msg*)

Restart the device server.

Returns **success** : {'ok', 'fail'}

Whether scheduling the restart succeeded.

Examples

```
?restart
!restart ok
```

request_sensor_list(*req*, *msg*)

Request the list of sensors.

The list of sensors is sent as a sequence of #sensor-list informs.

Parameters **name** : str, optional

Name of the sensor to list (the default is to list all sensors). If name starts and ends with '/' it is treated as a regular expression and all sensors whose names contain the regular expression are returned.

Informs **name** : str

The name of the sensor being described.

description : str

Description of the named sensor.

units : str

Units for the value of the named sensor.

type : str

Type of the named sensor.

params : list of str, optional

Additional sensor parameters (type dependent). For integer and float sensors the additional parameters are the minimum and maximum sensor value. For discrete sensors the additional parameters are the allowed values. For all other types no additional parameters are sent.

Returns **success** : {'ok', 'fail'}

Whether sending the sensor list succeeded.

informs : int

Number of #sensor-list inform messages sent.

Examples

```
?sensor-list
#sensor-list psu.voltage PSU\_voltage. V float 0.0 5.0
#sensor-list cpu.status CPU\_status. \@ discrete on off error
...
!sensor-list ok 5

?sensor-list cpu.power.on
#sensor-list cpu.power.on Whether\_CPU\_hase\_power. \@ boolean
!sensor-list ok 1

?sensor-list /voltage/
#sensor-list psu.voltage PSU\_voltage. V float 0.0 5.0
#sensor-list cpu.voltage CPU\_voltage. V float 0.0 3.0
!sensor-list ok 2
```

request_sensor_sampling(*req*, *msg*)

Configure or query the way a sensor is sampled.

Sampled values are reported asynchronously using the #sensor-status message.

Parameters **names** : str

One or more names of sensors whose sampling strategy will be queried or configured. If specifying multiple sensors, these must be provided as a comma-separated list. A query can only be done on a single sensor. However, configuration can be done on many sensors with a single request, as long as they all use the same strategy. Note: prior to KATCP v5.1 only a single sensor could be configured. Multiple sensors are only allowed if the device server sets the protocol version to KATCP v5.1 or higher and enables the BULK_SET_SENSOR_SAMPLING flag in its PROTOCOL_INFO class attribute.

strategy : {'none', 'auto', 'event', 'differential', 'differential-rate',

'period', 'event-rate'}, optional

Type of strategy to use to report the sensor value. The differential strategy types may only be used with integer or float sensors. If this parameter is supplied, it sets the new strategy.

params : list of str, optional

Additional strategy parameters (dependent on the strategy type). For the differential strategy, the parameter is an integer or float giving the amount by which the sensor value may change before an updated value is sent. For the period strategy, the parameter is the sampling period in float seconds. The event strategy has no parameters. Note that this has changed from KATCPv4. For the event-rate strategy, a minimum period between updates and a maximum period between updates (both in float seconds) must be given. If the event occurs more than once within the minimum period, only one update will occur. Whether or not the event occurs, the sensor value will be updated at least once per maximum period. For the differential-rate strategy there are 3 parameters. The first is the same as the differential strategy parameter. The second and third are the minimum and maximum periods, respectively, as with the event-rate strategy.

Informs timestamp : float

Timestamp of the sensor reading in seconds since the Unix epoch, or milliseconds for katcp versions <= 4.

count : {1}

Number of sensors described in this #sensor-status inform. Will always be one. It exists to keep this inform compatible with #sensor-value.

name : str

Name of the sensor whose value is being reported.

value : object

Value of the named sensor. Type depends on the type of the sensor.

Returns success : {'ok', 'fail'}

Whether the sensor-sampling request succeeded.

names : str

Name(s) of the sensor queried or configured. If multiple sensors, this will be a comma-separated list.

strategy : {'none', 'auto', 'event', 'differential', 'differential-rate',

'period', 'event-rate'}.

Name of the new or current sampling strategy for the sensor(s).

params : list of str

Additional strategy parameters (see description under Parameters).

Examples :

——— :

:: :

?sensor-sampling cpu.power.on !sensor-sampling ok cpu.power.on none

?sensor-sampling cpu.power.on period 0.5 #sensor-status 1244631611.415231 1 cpu.power.on nominal 1 !sensor-sampling ok cpu.power.on period 0.5

if BULK_SET_SENSOR_SAMPLING is enabled then:

?sensor-sampling cpu.power.on,fan.speed !sensor-sampling fail Cannot_query_multiple_sensors

?sensor-sampling cpu.power.on,fan.speed period 0.5 #sensor-status 1244631611.415231 1 cpu.power.on nominal 1 #sensor-status 1244631611.415200 1 fan.speed nominal 10.0 !sensor-sampling ok cpu.power.on,fan.speed period 0.5

request_sensor_sampling_clear(*req*, *msg*)

Set all sampling strategies for this client to none.

Returns **success** : {'ok', 'fail'}

Whether sending the list of devices succeeded.

Examples

?sensor-sampling-clear !sensor-sampling-clear ok

request_sensor_value(*req*, *msg*)

Request the value of a sensor or sensors.

A list of sensor values as a sequence of #sensor-value informs.

Parameters **name** : str, optional

Name of the sensor to poll (the default is to send values for all sensors). If name starts and ends with '/' it is treated as a regular expression and all sensors whose names contain the regular expression are returned.

Informs **timestamp** : float

Timestamp of the sensor reading in seconds since the Unix epoch, or milliseconds for katcp versions <= 4.

count : {1}

Number of sensors described in this #sensor-value inform. Will always be one. It exists to keep this inform compatible with #sensor-status.

name : str

Name of the sensor whose value is being reported.

value : object

Value of the named sensor. Type depends on the type of the sensor.

Returns **success** : {'ok', 'fail'}

Whether sending the list of values succeeded.

informs : int

Number of #sensor-value inform messages sent.

Examples

```
?sensor-value
#sensor-value 1244631611.415231 1 psu.voltage nominal 4.5
#sensor-value 1244631611.415200 1 cpu.status nominal off
...
!sensor-value ok 5

?sensor-value cpu.power.on
#sensor-value 1244631611.415231 1 cpu.power.on nominal 0
!sensor-value ok 1
```

request_version_list(*req*, *msg*)

Request the list of versions of roles and subcomponents.

Informs **name** : str

Name of the role or component.

version : str

A string identifying the version of the component. Individual components may define the structure of this argument as they choose. In the absence of other information clients should treat it as an opaque string.

build_state_or_serial_number : str

A unique identifier for a particular instance of a component. This should change whenever the component is replaced or updated.

Returns **success** : {'ok', 'fail'}

Whether sending the version list succeeded.

informs : int

Number of #version-list inform messages sent.

Examples

```
?version-list
#version-list katcp-protocol 5.0-MI
#version-list katcp-library katcp-python-0.4 katcp-python-0.4.1-py2
#version-list katcp-device foodevice-1.0 foodevice-1.0.0rc1
!version-list ok 3
```

request_watchdog(*req*, *msg*)

Check that the server is still alive.

Returns **success** : {'ok'}

Examples

```
?watchdog
!watchdog ok
```

running()

Whether the server is running.

setDaemon(*daemonic*)
Set daemonic state of the managed ioloop thread to True / False

Calling this method for a non-managed ioloop has no effect. Must be called before start(), or it will also have no effect

set_concurrency_options(*thread_safe=True*, *handler_thread=True*)
Set concurrency options for this device server. Must be called before *start()*.

Parameters **thread_safe** : bool

If True, make the server public methods thread safe. Incurs performance overhead.

handler_thread : bool

Can only be set if *thread_safe* is True. Handle all requests (even from different clients) in a separate, single, request-handling thread. Blocking request handlers will prevent the server from handling new requests from any client, but sensor strategies should still function. This more or less mimics the behaviour of a server in library versions before 0.6.0.

set_ioloop(*ioloop=None*)
Set the tornado IOLoop to use.

Sets the tornado.ioloop.IOLoop instance to use, defaulting to IOLoop.current(). If set_ioloop() is never called the IOLoop is started in a new thread, and will be stopped if self.stop() is called.

Notes

Must be called before start() is called.

set_restart_queue(*restart_queue*)
Set the restart queue.

When the device server should be restarted, it will be added to the queue.

Parameters **restart_queue** : Queue.Queue object

The queue to add the device server to when it should be restarted.

setup_sensors()
Populate the dictionary of sensors.

Unimplemented by default – subclasses should add their sensors here or pass if there are no sensors.

Examples

```
>>> class MyDevice(DeviceServer):
...     def setup_sensors(self):
...         self.add_sensor(Sensor(...))
...         self.add_sensor(Sensor(...))
...
```

start(*timeout=None*)
Start the server in a new thread.

Parameters **timeout** : float or None, optional

Time in seconds to wait for server thread to start.

stop(*timeout=1.0*)
Stop a running server (from another thread).

Parameters **timeout** : float, optional

Seconds to wait for server to have *started*.

Returns **stopped** : thread-safe Future

Resolves when the server is stopped

sync_with_ioloop(*timeout=None*)
Block for ioloop to complete a loop if called from another thread.

Returns a future if called from inside the ioloop.

Raises concurrent.futures.TimeoutError if timed out while blocking.

version()
Return a version string of the form type-major.minor.

wait_running(*timeout=None*)
Wait until the server is running

DeviceServer

class katcp.**DeviceServer**(**args*, ***kwargs*)
Implements some standard messages on top of DeviceServerBase.

Inform messages handled are:

- version (sent on connect)
- build-state (sent on connect)
- log (via self.log.warn(...), etc)
- disconnect
- client-connected

Requests handled are:

- halt
- help
- log-level
- restart[1]
- client-list
- sensor-list
- sensor-sampling
- sensor-value
- watchdog
- version-list (only standard in KATCP v5 or later)

[1] Restart relies on .set_restart_queue() being used to register a restart queue with the device. When the device needs to be restarted, it will be added to the restart queue. The queue should be a Python Queue.Queue object without a maximum size.

- **request-timeout-hint (pre-standard only if protocol flags indicates** timeout hints, supported for KATCP v5.1 or later)
- sensor-sampling-clear (non-standard)

Unhandled standard requests are:

- configure
- mode

Subclasses can define the tuple VERSION_INFO to set the interface name, major and minor version numbers. The BUILD_INFO tuple can be defined to give a string describing a particular interface instance and may have a fourth element containing additional version information (e.g. rc1).

Subclasses may manipulate the versions returned by the ?version-list command by editing .extra_versions which is a dictionary mapping role or component names to (version, build_state_or_serial_no) tuples. The build_state_or_serial_no may be None.

Subclasses must override the .setup_sensors() method. If they have no sensors to register, the method should just be a pass.

Methods

DeviceServer.add_sensor(sensor)	Add a sensor to the device.
DeviceServer.build_state()	Return build state string of the form name-major.minor[(a|b|rc)n].
DeviceServer.clear_strategies(client_conn[, ...])	Clear the sensor strategies of a client connection.
DeviceServer.create_exception_reply_and_log(...)	
DeviceServer.create_log_inform(level_name, ...)	Create a katcp logging inform message.
DeviceServer.get_sensor(sensor_name)	Fetch the sensor with the given name.
DeviceServer.get_sensors()	Fetch a list of all sensors.
DeviceServer.handle_inform(connection, msg)	Dispatch an inform message to the appropriate method.
DeviceServer.handle_message(client_conn, msg)	Handle messages of all types from clients.
DeviceServer.handle_reply(connection, msg)	Dispatch a reply message to the appropriate method.
DeviceServer.handle_request(connection, msg)	Dispatch a request message to the appropriate method.
DeviceServer.has_sensor(sensor_name)	Whether the sensor with specified name is known.
DeviceServer.inform(connection, msg)	Send an inform message to a particular client.
DeviceServer.join([timeout])	Rejoin the server thread.
DeviceServer.mass_inform(msg)	Send an inform message to all clients.
DeviceServer.next()	
DeviceServer.on_client_connect(**kwargs)	Inform client of build state and version on connect.
DeviceServer.on_client_disconnect(...)	Inform client it is about to be disconnected.
DeviceServer.on_message(client_conn, msg)	Dummy implementation of on_message required by KATCPServer.
DeviceServer.remove_sensor(sensor)	Remove a sensor from the device.
DeviceServer.reply(connection, reply, orig_req)	Send an asynchronous reply to an earlier request.

Continued on next page

Table 6 – continued from previous page

DeviceServer.reply_inform(connection, ...)	Send an inform as part of the reply to an earlier request.
DeviceServer.request_client_list(req, msg)	Request the list of connected clients.
DeviceServer.request_halt(req, msg)	Halt the device server.
DeviceServer.request_help(req, msg)	Return help on the available requests.
DeviceServer.request_log_level(req, msg)	Query or set the current logging level.
DeviceServer.request_request_timeout_hint(req, msg)	Return timeout hints for requests
DeviceServer.request_restart(req, msg)	Restart the device server.
DeviceServer.request_sensor_list(req, msg)	Request the list of sensors.
DeviceServer.request_sensor_sampling(req, msg)	Configure or query the way a sensor is sampled.
DeviceServer.request_sensor_sampling_clear()	Set all sampling strategies for this client to none.
DeviceServer.request_sensor_value(req, msg)	Request the value of a sensor or sensors.
DeviceServer.request_version_list(req, msg)	Request the list of versions of roles and subcomponents.
DeviceServer.request_watchdog(req, msg)	Check that the server is still alive.
DeviceServer.running()	Whether the server is running.
DeviceServer.setDaemon(daemonic)	Set daemonic state of the managed ioloop thread to True / False
DeviceServer.set_concurrency_options([...])	Set concurrency options for this device server.
DeviceServer.set_ioloop([ioloop])	Set the tornado IOLoop to use.
DeviceServer.set_restart_queue(restart_queue)	Set the restart queue.
DeviceServer.setup_sensors()	Populate the dictionary of sensors.
DeviceServer.start([timeout])	Start the server in a new thread.
DeviceServer.stop([timeout])	Stop a running server (from another thread).
DeviceServer.sync_with_ioloop([timeout])	Block for ioloop to complete a loop if called from another thread.
DeviceServer.version()	Return a version string of the form type-major.minor.
DeviceServer.wait_running([timeout])	Wait until the server is running

add_sensor(*sensor*)
Add a sensor to the device.

Usually called inside .setup_sensors() but may be called from elsewhere.

Parameters **sensor** : Sensor object

The sensor object to register with the device server.

build_state()
Return build state string of the form name-major.minor[(a|b|rc)n].

clear_strategies(*client_conn*, *remove_client=False*)
Clear the sensor strategies of a client connection.

Parameters **client_connection** : ClientConnection instance

The connection that should have its sampling strategies cleared

remove_client : bool, optional

Remove the client connection from the strategies data-structure. Useful for clients that disconnect.

create_log_inform(*level_name*, *msg*, *name*, *timestamp=None*)
Create a katcp logging inform message.

Usually this will be called from inside a DeviceLogger object, but it is also used by the methods in this class when errors need to be reported to the client.

get_sensor(*sensor_name*)
Fetch the sensor with the given name.

Parameters **sensor_name** : str

Name of the sensor to retrieve.

Returns **sensor** : Sensor object

The sensor with the given name.

get_sensors()
Fetch a list of all sensors.

Returns **sensors** : list of Sensor objects

The list of sensors registered with the device server.

handle_inform(*connection*, *msg*)
Dispatch an inform message to the appropriate method.

Parameters **connection** : ClientConnection object

The client connection the message was from.

msg : Message object

The inform message to process.

handle_message(*client_conn*, *msg*)
Handle messages of all types from clients.

Parameters **client_conn** : ClientConnection object

The client connection the message was from.

msg : Message object

The message to process.

handle_reply(*connection*, *msg*)
Dispatch a reply message to the appropriate method.

Parameters **connection** : ClientConnection object

The client connection the message was from.

msg : Message object

The reply message to process.

handle_request(*connection*, *msg*)
Dispatch a request message to the appropriate method.

Parameters **connection** : ClientConnection object

The client connection the message was from.

msg : Message object

The request message to process.

Returns **done_future** : Future or None

Returns Future for async request handlers that will resolve when done, or None for sync request handlers once they have completed.

has_sensor(*sensor_name*)
Whether the sensor with specified name is known.

inform(*connection*, *msg*)
Send an inform message to a particular client.

Should only be used for asynchronous informs. Informs that are part of the response to a request should use *reply_inform()* so that the message identifier from the original request can be attached to the inform.

Parameters **connection** : ClientConnection object

The client to send the message to.

msg : Message object

The inform message to send.

join(*timeout=None*)
Rejoin the server thread.

Parameters **timeout** : float or None, optional

Time in seconds to wait for the thread to finish.

mass_inform(*msg*)
Send an inform message to all clients.

Parameters **msg** : Message object

The inform message to send.

on_client_connect(***kwargs*)
Inform client of build state and version on connect.

Parameters **client_conn** : ClientConnection object

The client connection that has been successfully established.

Returns **Future that resolves when the device is ready to accept messages.** :

on_client_disconnect(*client_conn*, *msg*, *connection_valid*)
Inform client it is about to be disconnected.

Parameters **client_conn** : ClientConnection object

The client connection being disconnected.

msg : str

Reason client is being disconnected.

connection_valid : bool

True if connection is still open for sending, False otherwise.

Returns **Future that resolves when the client connection can be closed.** :

on_message(*client_conn*, *msg*)
Dummy implementation of on_message required by KATCPServer.

Will be replaced by a handler with the appropriate concurrency semantics when set_concurrency_options is called (defaults are set in __init__()).

remove_sensor(*sensor*)
Remove a sensor from the device.

Also deregisters all clients observing the sensor.

Parameters **sensor** : Sensor object or name string

The sensor to remove from the device server.

reply(*connection*, *reply*, *orig_req*)
Send an asynchronous reply to an earlier request.

Parameters **connection** : ClientConnection object

The client to send the reply to.

reply : Message object

The reply message to send.

orig_req : Message object

The request message being replied to. The reply message's id is overridden with the id from orig_req before the reply is sent.

reply_inform(*connection*, *inform*, *orig_req*)
Send an inform as part of the reply to an earlier request.

Parameters **connection** : ClientConnection object

The client to send the inform to.

inform : Message object

The inform message to send.

orig_req : Message object

The request message being replied to. The inform message's id is overridden with the id from orig_req before the inform is sent.

request_client_list(*req*, *msg*)
Request the list of connected clients.

The list of clients is sent as a sequence of #client-list informs.

Informs **addr** : str

The address of the client as host:port with host in dotted quad notation. If the address of the client could not be determined (because, for example, the client disconnected suddenly) then a unique string representing the client is sent instead.

Returns **success** : {'ok', 'fail'}

Whether sending the client list succeeded.

informs : int

Number of #client-list inform messages sent.

Examples

```
?client-list
#client-list 127.0.0.1:53600
!client-list ok 1
```

request_halt (*req*, *msg*)
Halt the device server.

Returns **success** : {'ok', 'fail'}

Whether scheduling the halt succeeded.

Examples

```
?halt
!halt ok
```

request_help (*req*, *msg*)
Return help on the available requests.

Return a description of the available requests using a sequence of #help informs.

Parameters **request** : str, optional

The name of the request to return help for (the default is to return help for all requests).

Informs **request** : str

The name of a request.

description : str

Documentation for the named request.

Returns **success** : {'ok', 'fail'}

Whether sending the help succeeded.

informs : int

Number of #help inform messages sent.

Examples

```
?help
#help halt ...description...
#help help ...description...
...
!help ok 5

?help halt
#help halt ...description...
!help ok 1
```

request_log_level (*req*, *msg*)
Query or set the current logging level.

Parameters **level** : {'all', 'trace', 'debug', 'info', 'warn', 'error', 'fatal', 'off'}, optional

Name of the logging level to set the device server to (the default is to leave the log level unchanged).

Returns **success** : {'ok', 'fail'}

Whether the request succeeded.

level : {'all', 'trace', 'debug', 'info', 'warn', 'error', 'fatal', 'off'}

The log level after processing the request.

Examples

```
?log-level
!log-level ok warn

?log-level info
!log-level ok info
```

request_request_timeout_hint(*req*, *msg*)

Return timeout hints for requests

KATCP requests should generally take less than 5s to complete, but some requests are unavoidably slow. This results in spurious client timeout errors. This request provides timeout hints that clients can use to select suitable request timeouts.

Parameters **request** : str, optional

The name of the request to return a timeout hint for (the default is to return hints for all requests that have timeout hints). Returns one inform per request. Must be an existing request if specified.

Informs **request** : str

The name of the request.

suggested_timeout : float

Suggested request timeout in seconds for the request. If *suggested_timeout* is zero (0), no timeout hint is available.

Returns **success** : {'ok', 'fail'}

Whether sending the help succeeded.

informs : int

Number of #request-timeout-hint inform messages sent.

Notes

?request-timeout-hint without a parameter will only return informs for requests that have specific timeout hints, so it will most probably be a subset of all the requests, or even no informs at all.

Examples

```
?request-timeout-hint
#request-timeout-hint halt 5
#request-timeout-hint very-slow-request 500
...
!request-timeout-hint ok 5

?request-timeout-hint moderately-slow-request
#request-timeout-hint moderately-slow-request 20
!request-timeout-hint ok 1
```

request_restart(*req*, *msg*)
Restart the device server.

Returns **success** : {'ok', 'fail'}

Whether scheduling the restart succeeded.

Examples

```
?restart
!restart ok
```

request_sensor_list(*req*, *msg*)
Request the list of sensors.

The list of sensors is sent as a sequence of #sensor-list informs.

Parameters **name** : str, optional

Name of the sensor to list (the default is to list all sensors). If name starts and ends with '/' it is treated as a regular expression and all sensors whose names contain the regular expression are returned.

Informs **name** : str

The name of the sensor being described.

description : str

Description of the named sensor.

units : str

Units for the value of the named sensor.

type : str

Type of the named sensor.

params : list of str, optional

Additional sensor parameters (type dependent). For integer and float sensors the additional parameters are the minimum and maximum sensor value. For discrete sensors the additional parameters are the allowed values. For all other types no additional parameters are sent.

Returns **success** : {'ok', 'fail'}

Whether sending the sensor list succeeded.

informs : int

Number of #sensor-list inform messages sent.

Examples

```
?sensor-list
#sensor-list psu.voltage PSU\_voltage. V float 0.0 5.0
#sensor-list cpu.status CPU\_status. \@ discrete on off error
...
!sensor-list ok 5

?sensor-list cpu.power.on
#sensor-list cpu.power.on Whether\_CPU\_hase\_power. \@ boolean
!sensor-list ok 1

?sensor-list /voltage/
#sensor-list psu.voltage PSU\_voltage. V float 0.0 5.0
#sensor-list cpu.voltage CPU\_voltage. V float 0.0 3.0
!sensor-list ok 2
```

request_sensor_sampling(*req*, *msg*)

Configure or query the way a sensor is sampled.

Sampled values are reported asynchronously using the #sensor-status message.

Parameters **names** : str

One or more names of sensors whose sampling strategy will be queried or configured. If specifying multiple sensors, these must be provided as a comma-separated list. A query can only be done on a single sensor. However, configuration can be done on many sensors with a single request, as long as they all use the same strategy. Note: prior to KATCP v5.1 only a single sensor could be configured. Multiple sensors are only allowed if the device server sets the protocol version to KATCP v5.1 or higher and enables the BULK_SET_SENSOR_SAMPLING flag in its PROTOCOL_INFO class attribute.

strategy : {'none', 'auto', 'event', 'differential', 'differential-rate',

'period', 'event-rate'}, optional

Type of strategy to use to report the sensor value. The differential strategy types may only be used with integer or float sensors. If this parameter is supplied, it sets the new strategy.

params : list of str, optional

Additional strategy parameters (dependent on the strategy type). For the differential strategy, the parameter is an integer or float giving the amount by which the sensor value may change before an updated value is sent. For the period strategy, the parameter is the sampling period in float seconds. The event strategy has no parameters. Note that this has changed from KATCPv4. For the event-rate strategy, a minimum period between updates and a maximum period between updates (both in float seconds) must be given. If the event occurs more than once within the minimum period, only one update will occur. Whether or not the event occurs, the sensor value will be updated at least once per maximum period. For the differential-rate strategy there are 3 parameters. The first is the same as the differential strategy parameter. The second and third are the minimum and maximum periods, respectively, as with the event-rate strategy.

Informs **timestamp** : float

Timestamp of the sensor reading in seconds since the Unix epoch, or milliseconds for katcp versions <= 4.

count : {1}

Number of sensors described in this #sensor-status inform. Will always be one. It exists to keep this inform compatible with #sensor-value.

name : str

Name of the sensor whose value is being reported.

value : object

Value of the named sensor. Type depends on the type of the sensor.

Returns **success** : {'ok', 'fail'}

Whether the sensor-sampling request succeeded.

names : str

Name(s) of the sensor queried or configured. If multiple sensors, this will be a comma-separated list.

strategy : {'none', 'auto', 'event', 'differential', 'differential-rate',

'period', 'event-rate'}.

Name of the new or current sampling strategy for the sensor(s).

params : list of str

Additional strategy parameters (see description under Parameters).

Examples :

——— :

:: :

?sensor-sampling cpu.power.on !sensor-sampling ok cpu.power.on none

?sensor-sampling cpu.power.on period 0.5 #sensor-status 1244631611.415231 1 cpu.power.on nominal 1 !sensor-sampling ok cpu.power.on period 0.5

if BULK_SET_SENSOR_SAMPLING is enabled then:

?sensor-sampling cpu.power.on,fan.speed !sensor-sampling fail Cannot_query_multiple_sensors

?sensor-sampling cpu.power.on,fan.speed period 0.5 #sensor-status 1244631611.415231 1 cpu.power.on nominal 1 #sensor-status 1244631611.415200 1 fan.speed nominal 10.0 !sensor-sampling ok cpu.power.on,fan.speed period 0.5

request_sensor_sampling_clear(*req*, *msg*)

Set all sampling strategies for this client to none.

Returns **success** : {'ok', 'fail'}

Whether sending the list of devices succeeded.

Examples

?sensor-sampling-clear !sensor-sampling-clear ok

request_sensor_value(*req*, *msg*)

Request the value of a sensor or sensors.

A list of sensor values as a sequence of #sensor-value informs.

Parameters **name** : str, optional

Name of the sensor to poll (the default is to send values for all sensors). If name starts and ends with '/' it is treated as a regular expression and all sensors whose names contain the regular expression are returned.

Informs **timestamp** : float

Timestamp of the sensor reading in seconds since the Unix epoch, or milliseconds for katcp versions <= 4.

count : {1}

Number of sensors described in this #sensor-value inform. Will always be one. It exists to keep this inform compatible with #sensor-status.

name : str

Name of the sensor whose value is being reported.

value : object

Value of the named sensor. Type depends on the type of the sensor.

Returns **success** : {'ok', 'fail'}

Whether sending the list of values succeeded.

informs : int

Number of #sensor-value inform messages sent.

Examples

```
?sensor-value
#sensor-value 1244631611.415231 1 psu.voltage nominal 4.5
#sensor-value 1244631611.415200 1 cpu.status nominal off
...
!sensor-value ok 5

?sensor-value cpu.power.on
#sensor-value 1244631611.415231 1 cpu.power.on nominal 0
!sensor-value ok 1
```

request_version_list(*req*, *msg*)

Request the list of versions of roles and subcomponents.

Informs **name** : str

Name of the role or component.

version : str

A string identifying the version of the component. Individual components may define the structure of this argument as they choose. In the absence of other information clients should treat it as an opaque string.

build_state_or_serial_number : str

A unique identifier for a particular instance of a component. This should change whenever the component is replaced or updated.

Returns **success** : {'ok', 'fail'}

Whether sending the version list succeeded.

informs : int

Number of #version-list inform messages sent.

Examples

```
?version-list
#version-list katcp-protocol 5.0-MI
#version-list katcp-library katcp-python-0.4 katcp-python-0.4.1-py2
#version-list katcp-device foodevice-1.0 foodevice-1.0.0rc1
!version-list ok 3
```

request_watchdog(*req*, *msg*)
Check that the server is still alive.

Returns **success** : {'ok'}

Examples

```
?watchdog
!watchdog ok
```

running()
Whether the server is running.

setDaemon(*daemonic*)
Set daemonic state of the managed ioloop thread to True / False

Calling this method for a non-managed ioloop has no effect. Must be called before start(), or it will also have no effect

set_concurrency_options(*thread_safe=True*, *handler_thread=True*)
Set concurrency options for this device server. Must be called before *start()*.

Parameters **thread_safe** : bool

If True, make the server public methods thread safe. Incurs performance overhead.

handler_thread : bool

Can only be set if *thread_safe* is True. Handle all requests (even from different clients) in a separate, single, request-handling thread. Blocking request handlers will prevent the server from handling new requests from any client, but sensor strategies should still function. This more or less mimics the behaviour of a server in library versions before 0.6.0.

set_ioloop(*ioloop=None*)
Set the tornado IOLoop to use.

Sets the tornado.ioloop.IOLoop instance to use, defaulting to IOLoop.current(). If set_ioloop() is never called the IOLoop is started in a new thread, and will be stopped if self.stop() is called.

Notes

Must be called before start() is called.

set_restart_queue(*restart_queue*)
Set the restart queue.

When the device server should be restarted, it will be added to the queue.

Parameters **restart_queue** : Queue.Queue object

The queue to add the device server to when it should be restarted.

setup_sensors()
Populate the dictionary of sensors.

Unimplemented by default – subclasses should add their sensors here or pass if there are no sensors.

Examples

```
>>> class MyDevice(DeviceServer):
...     def setup_sensors(self):
...         self.add_sensor(Sensor(...))
...         self.add_sensor(Sensor(...))
...
```

start(*timeout=None*)
Start the server in a new thread.

Parameters **timeout** : float or None, optional

Time in seconds to wait for server thread to start.

stop(*timeout=1.0*)
Stop a running server (from another thread).

Parameters **timeout** : float, optional

Seconds to wait for server to have *started*.

Returns **stopped** : thread-safe Future

Resolves when the server is stopped

sync_with_ioloop(*timeout=None*)
Block for ioloop to complete a loop if called from another thread.

Returns a future if called from inside the ioloop.

Raises concurrent.futures.TimeoutError if timed out while blocking.

version()
Return a version string of the form type-major.minor.

wait_running(*timeout=None*)
Wait until the server is running

DeviceServerBase

class katcp.**DeviceServerBase**(*host*, *port*, *tb_limit=20*, *logger=<logging.Logger object>*)
Base class for device servers.

Subclasses should add .request_* methods for dealing with request messages. These methods each take the client request connection and msg objects as arguments and should return the reply message or raise an exception as a result.

Subclasses can also add .inform_* and reply_* methods to handle those types of messages.

Should a subclass need to generate inform messages it should do so using either the .inform() or .mass_inform() methods.

Finally, this class should probably not be subclassed directly but rather via subclassing DeviceServer itself which implements common .request_* methods.

Parameters **host** : str

Host to listen on.

port : int

Port to listen on.

tb_limit : int, optional

Maximum number of stack frames to send in error tracebacks.

logger : logging.Logger object, optional

Logger to log messages to.

Methods

`DeviceServerBase.create_exception_reply_and_log(...)`	
`DeviceServerBase.create_log_inform`(...[, ...])	Create a katcp logging inform message.
`DeviceServerBase.handle_inform`(connection, msg)	Dispatch an inform message to the appropriate method.
`DeviceServerBase.handle_message`(client_conn, msg)	Handle messages of all types from clients.
`DeviceServerBase.handle_reply`(connection, msg)	Dispatch a reply message to the appropriate method.
`DeviceServerBase.handle_request`(connection, msg)	Dispatch a request message to the appropriate method.
`DeviceServerBase.inform`(connection, msg)	Send an inform message to a particular client.
`DeviceServerBase.join`([timeout])	Rejoin the server thread.
`DeviceServerBase.mass_inform`(msg)	Send an inform message to all clients.
`DeviceServerBase.next()`	
`DeviceServerBase.on_client_connect`(**kwargs)	Called after client connection is established.
`DeviceServerBase.on_client_disconnect`(**kwargs)	Called before a client connection is closed.
`DeviceServerBase.on_message`(client_conn, msg)	Dummy implementation of on_message required by KATCPServer.
`DeviceServerBase.reply`(connection, reply, ...)	Send an asynchronous reply to an earlier request.
`DeviceServerBase.reply_inform`(connection, ...)	Send an inform as part of the reply to an earlier request.
`DeviceServerBase.running`()	Whether the server is running.
`DeviceServerBase.setDaemon`(daemonic)	Set daemonic state of the managed ioloop thread to True / False

Continued on next page

Table 7 – continued from previous page

DeviceServerBase.set_concurrency_options([...])	Set concurrency options for this device server.
DeviceServerBase.set_ioloop([ioloop])	Set the tornado IOLoop to use.
DeviceServerBase.start([timeout])	Start the server in a new thread.
DeviceServerBase.stop([timeout])	Stop a running server (from another thread).
DeviceServerBase.sync_with_ioloop([timeout])	Block for ioloop to complete a loop if called from another thread.
DeviceServerBase.wait_running([timeout])	Wait until the server is running

create_log_inform(*level_name*, *msg*, *name*, *timestamp=None*)
Create a katcp logging inform message.

Usually this will be called from inside a DeviceLogger object, but it is also used by the methods in this class when errors need to be reported to the client.

handle_inform(*connection*, *msg*)
Dispatch an inform message to the appropriate method.

Parameters **connection** : ClientConnection object

The client connection the message was from.

msg : Message object

The inform message to process.

handle_message(*client_conn*, *msg*)
Handle messages of all types from clients.

Parameters **client_conn** : ClientConnection object

The client connection the message was from.

msg : Message object

The message to process.

handle_reply(*connection*, *msg*)
Dispatch a reply message to the appropriate method.

Parameters **connection** : ClientConnection object

The client connection the message was from.

msg : Message object

The reply message to process.

handle_request(*connection*, *msg*)
Dispatch a request message to the appropriate method.

Parameters **connection** : ClientConnection object

The client connection the message was from.

msg : Message object

The request message to process.

Returns **done_future** : Future or None

Returns Future for async request handlers that will resolve when done, or None for sync request handlers once they have completed.

inform(*connection*, *msg*)
Send an inform message to a particular client.

Should only be used for asynchronous informs. Informs that are part of the response to a request should use *reply_inform()* so that the message identifier from the original request can be attached to the inform.

Parameters **connection** : ClientConnection object

The client to send the message to.

msg : Message object

The inform message to send.

join(*timeout=None*)
Rejoin the server thread.

Parameters **timeout** : float or None, optional

Time in seconds to wait for the thread to finish.

mass_inform(*msg*)
Send an inform message to all clients.

Parameters **msg** : Message object

The inform message to send.

on_client_connect(***kwargs*)
Called after client connection is established.

Subclasses should override if they wish to send clients message or perform house-keeping at this point.

Parameters **conn** : ClientConnection object

The client connection that has been successfully established.

Returns **Future that resolves when the device is ready to accept messages.** :

on_client_disconnect(***kwargs*)
Called before a client connection is closed.

Subclasses should override if they wish to send clients message or perform house-keeping at this point. The server cannot guarantee this will be called (for example, the client might drop the connection). The message parameter contains the reason for the disconnection.

Parameters **conn** : ClientConnection object

Client connection being disconnected.

msg : str

Reason client is being disconnected.

connection_valid : boolean

True if connection is still open for sending, False otherwise.

Returns **Future that resolves when the client connection can be closed.** :

on_message(*client_conn*, *msg*)
Dummy implementation of on_message required by KATCPServer.

Will be replaced by a handler with the appropriate concurrency semantics when set_concurrency_options is called (defaults are set in __init__()).

reply(*connection*, *reply*, *orig_req*)
Send an asynchronous reply to an earlier request.

Parameters **connection** : ClientConnection object

The client to send the reply to.

reply : Message object

The reply message to send.

orig_req : Message object

The request message being replied to. The reply message's id is overridden with the id from orig_req before the reply is sent.

reply_inform(*connection*, *inform*, *orig_req*)
Send an inform as part of the reply to an earlier request.

Parameters **connection** : ClientConnection object

The client to send the inform to.

inform : Message object

The inform message to send.

orig_req : Message object

The request message being replied to. The inform message's id is overridden with the id from orig_req before the inform is sent.

running()
Whether the server is running.

setDaemon(*daemonic*)
Set daemonic state of the managed ioloop thread to True / False

Calling this method for a non-managed ioloop has no effect. Must be called before start(), or it will also have no effect

set_concurrency_options(*thread_safe=True*, *handler_thread=True*)
Set concurrency options for this device server. Must be called before *start()*.

Parameters **thread_safe** : bool

If True, make the server public methods thread safe. Incurs performance overhead.

handler_thread : bool

Can only be set if *thread_safe* is True. Handle all requests (even from different clients) in a separate, single, request-handling thread. Blocking request handlers will prevent the server from handling new requests from any client, but sensor strategies should still function. This more or less mimics the behaviour of a server in library versions before 0.6.0.

set_ioloop(*ioloop=None*)
Set the tornado IOLoop to use.

Sets the tornado.ioloop.IOLoop instance to use, defaulting to IOLoop.current(). If set_ioloop() is never called the IOLoop is started in a new thread, and will be stopped if self.stop() is called.

Notes

Must be called before start() is called.

start(*timeout=None*)
Start the server in a new thread.

Parameters **timeout** : float or None, optional

Time in seconds to wait for server thread to start.

stop(*timeout=1.0*)
Stop a running server (from another thread).

Parameters **timeout** : float, optional

Seconds to wait for server to have *started*.

Returns **stopped** : thread-safe Future

Resolves when the server is stopped

sync_with_ioloop(*timeout=None*)
Block for ioloop to complete a loop if called from another thread.

Returns a future if called from inside the ioloop.

Raises concurrent.futures.TimeoutError if timed out while blocking.

wait_running(*timeout=None*)
Wait until the server is running

DeviceLogger

class katcp.**DeviceLogger**(*device_server*, *root_logger='root'*, *python_logger=None*)
Object for logging messages from a DeviceServer.

Log messages are logged at a particular level and under a particular name. Names use dotted notation to form a virtual hierarchy of loggers with the device.

Parameters **device_server** : DeviceServerBase object

The device server this logger should use for sending out logs.

root_logger : str

The name of the root logger.

Methods

DeviceLogger.debug(msg, *args, **kwargs)	Log a debug message.
DeviceLogger.error(msg, *args, **kwargs)	Log an error message.
DeviceLogger.fatal(msg, *args, **kwargs)	Log a fatal error message.
DeviceLogger.info(msg, *args, **kwargs)	Log an info message.
DeviceLogger.level_from_name(level_name)	Return the level constant for a given name.
DeviceLogger.level_name([level])	Return the name of the given level value.
DeviceLogger.log(level, msg, *args, **kwargs)	Log a message and inform all clients.
DeviceLogger.log_to_python(logger, msg)	Log a KATCP logging message to a Python logger.

Continued on next page

Table 8 – continued from previous page

DeviceLogger.next()	
DeviceLogger.set_log_level(level)	Set the logging level.
DeviceLogger.set_log_level_by_name(level_name)	Set the logging level using a level name.
DeviceLogger.trace(msg, *args, **kwargs)	Log a trace message.
DeviceLogger.warn(msg, *args, **kwargs)	Log an warning message.

debug(*msg*, **args*, ***kwargs*)
Log a debug message.

error(*msg*, **args*, ***kwargs*)
Log an error message.

fatal(*msg*, **args*, ***kwargs*)
Log a fatal error message.

info(*msg*, **args*, ***kwargs*)
Log an info message.

level_from_name(*level_name*)
Return the level constant for a given name.

If the *level_name* is not known, raise a ValueError.

Parameters **level_name** : str or bytes

The logging level name whose logging level constant to retrieve.

Returns **level** : logging level constant

The logging level constant associated with the name.

level_name(*level=None*)
Return the name of the given level value.

If level is None, return the name of the current level.

Parameters **level** : logging level constant

The logging level constant whose name to retrieve.

Returns **level_name** : str

The name of the logging level.

log(*level*, *msg*, **args*, ***kwargs*)
Log a message and inform all clients.

Parameters **level** : logging level constant

The level to log the message at.

msg : str

The text format for the log message.

args : list of objects

Arguments to pass to log format string. Final message text is created using: msg % args.

kwargs : additional keyword parameters

Allowed keywords are 'name' and 'timestamp'. The name is the name of the logger to log the message to. If not given the name defaults to the root logger. The timestamp is a float in seconds. If not given the timestamp defaults to the current time.

classmethod log_to_python(*logger*, *msg*)
Log a KATCP logging message to a Python logger.

Parameters **logger** : logging.Logger object

The Python logger to log the given message to.

msg : Message object

The #log message to create a log entry from.

set_log_level(*level*)
Set the logging level.

Parameters **level** : logging level constant

The value to set the logging level to.

set_log_level_by_name(*level_name*)
Set the logging level using a level name.

Parameters **level_name** : str or bytes

The name of the logging level.

trace(*msg*, **args*, ***kwargs*)
Log a trace message.

warn(*msg*, **args*, ***kwargs*)
Log an warning message.

Sensor

class katcp.**Sensor**(*sensor_type*, *name*, *description=None*, *units=None*, *params=None*, *default=None*, *initial_status=None*)
Instantiate a new sensor object.

Subclasses will usually pass in a fixed sensor_type which should be one of the sensor type constants. The list params if set will have its values formatted by the type formatter for the given sensor type.

Note: The LRU sensor type was deprecated in katcp 0.4.

Note: The ADDRESS sensor type was added in katcp 0.4.

Parameters **sensor_type** : Sensor type constant

The type of sensor.

name : str

The name of the sensor.

description : str, optional

A short description of the sensor.

units : str, optional

The units of the sensor value. May be the empty string if there are no applicable units.

params : list, optional

Additional parameters, dependent on the type of sensor:

- For `INTEGER` and `FLOAT` the list is optional. If provided, it should have two items, providing the minimum and maximum that define the range of the sensor value, respectively. The type of each item must be int, or float.
- For `DISCRETE` the list is required, and must contain all possible values the sensor may take. There must be at least one item. The type of each item must be str or bytes.
- For all other types, params should be omitted.

default : object, optional

An initial value for the sensor. By default this is determined by the sensor type. For `INTEGER` and `FLOAT` sensors, if no default is provided, but valid minimum and maximum parameters are, the default will be set to the minimum.

initial_status : int enum or None, optional

An initial status for the sensor. If None, defaults to Sensor.UNKNOWN. *initial_status* must be one of the keys in Sensor.STATUSES

Methods

Sensor.address(name[, description, unit, ...])	Instantiate a new IP address sensor object.
Sensor.attach(observer)	Attach an observer to this sensor.
Sensor.boolean(name[, description, unit, ...])	Instantiate a new boolean sensor object.
Sensor.detach(observer)	Detach an observer from this sensor.
Sensor.discrete(name[, description, unit, ...])	Instantiate a new discrete sensor object.
Sensor.float(name[, description, unit, ...])	Instantiate a new float sensor object.
Sensor.format_reading(reading[, major])	Format sensor reading as (timestamp, status, value) tuple of byte strings.
Sensor.integer(name[, description, unit, ...])	Instantiate a new integer sensor object.
Sensor.lru(name[, description, unit, ...])	Instantiate a new lru sensor object.
Sensor.notify(reading)	Notify all observers of changes to this sensor.
Sensor.parse_params(sensor_type, ...[, major])	Parse KATCP formatted parameters into Python values.
Sensor.parse_type(type_string)	Parse KATCP formatted type code into Sensor type constant.
Sensor.parse_value(s_value[, katcp_major])	Parse a value from a byte string.
Sensor.read()	Read the sensor and return a (timestamp, status, value) tuple.
Sensor.read_formatted([major])	Read the sensor and return a (timestamp, status, value) tuple.
Sensor.set(timestamp, status, value)	Set the current value of the sensor.
Sensor.set_formatted(raw_timestamp, ...[, major])	Set the current value of the sensor.
Sensor.set_value(value[, status, timestamp, ...])	Check and then set the value of the sensor.
Sensor.status()	Read the current sensor status.
Sensor.string(name[, description, unit, ...])	Instantiate a new string sensor object.
Sensor.timestamp(name[, description, unit, ...])	Instantiate a new timestamp sensor object.

Continued on next page

Table 9 – continued from previous page

Sensor.value()	Read the current sensor value.

classmethod address(*name*, *description=None*, *unit=''*, *default=None*, *initial_status=None*)
Instantiate a new IP address sensor object.

Parameters **name** : str

The name of the sensor.

description : str

A short description of the sensor.

unit : str

The units of the sensor value. May be the empty string if there are no applicable units.

default : (string, int)

An initial value for the sensor. Tuple containing (host, port). default is ("0.0.0.0", None)

initial_status : int enum or None

An initial status for the sensor. If None, defaults to Sensor.UNKNOWN. *initial_status* must be one of the keys in Sensor.STATUSES

attach(*observer*)
Attach an observer to this sensor.

The observer must support a call to observer.update(sensor, reading), where *sensor* is the sensor object and *reading* is a (timestamp, status, value) tuple for this update (matching the return value of the *read()* method).

Parameters **observer** : object

Object with an .update(sensor, reading) method that will be called when the sensor value is set

classmethod boolean(*name*, *description=None*, *unit=''*, *default=None*, *initial_status=None*)
Instantiate a new boolean sensor object.

Parameters **name** : str

The name of the sensor.

description : str

A short description of the sensor.

unit : str

The units of the sensor value. May be the empty string if there are no applicable units.

default : bool

An initial value for the sensor. Defaults to False.

initial_status : int enum or None

An initial status for the sensor. If None, defaults to Sensor.UNKNOWN. *initial_status* must be one of the keys in Sensor.STATUSES

detach(*observer*)
Detach an observer from this sensor.

Parameters **observer** : object

The observer to remove from the set of observers notified when the sensor value is set.

classmethod discrete(*name*, *description=None*, *unit=''*, *params=None*, *default=None*, *initial_status=None*)

Instantiate a new discrete sensor object.

Parameters **name** : str

The name of the sensor.

description : str

A short description of the sensor.

unit : str

The units of the sensor value. May be the empty string if there are no applicable units.

params : [str]

Sequence of all allowable discrete sensor states

default : str

An initial value for the sensor. Defaults to the first item of params

initial_status : int enum or None

An initial status for the sensor. If None, defaults to Sensor.UNKNOWN. *initial_status* must be one of the keys in Sensor.STATUSES

classmethod float(*name*, *description=None*, *unit=''*, *params=None*, *default=None*, *initial_status=None*)

Instantiate a new float sensor object.

Parameters **name** : str

The name of the sensor.

description : str

A short description of the sensor.

unit : str

The units of the sensor value. May be the empty string if there are no applicable units.

params : list

[min, max] – minimum and maximum values of the sensor

default : float

An initial value for the sensor. Defaults to 0.0.

initial_status : int enum or None

An initial status for the sensor. If None, defaults to Sensor.UNKNOWN. *initial_status* must be one of the keys in Sensor.STATUSES

format_reading(*reading*, *major=5*)

Format sensor reading as (timestamp, status, value) tuple of byte strings.

All values are strings formatted as specified in the Sensor Type Formats in the katcp specification.

Parameters **reading** : `Reading` object

Sensor reading as returned by *`read()`*

major : int

Major version of KATCP to use when interpreting types. Defaults to latest implemented KATCP version.

Returns **timestamp** : bytes

KATCP formatted timestamp byte string

status : bytes

KATCP formatted sensor status byte string

value : bytes

KATCP formatted sensor value byte string

Notes

Should only be used for a reading obtained from the same sensor.

classmethod integer(*name*, *description=None*, *unit=''*, *params=None*, *default=None*, *initial_status=None*)

Instantiate a new integer sensor object.

Parameters **name** : str

The name of the sensor.

description : str

A short description of the sensor.

unit : str

The units of the sensor value. May be the empty string if there are no applicable units.

params : list

[min, max] – minimum and maximum values of the sensor

default : int

An initial value for the sensor. Defaults to 0.

initial_status : int enum or None

An initial status for the sensor. If None, defaults to Sensor.UNKNOWN. *initial_status* must be one of the keys in Sensor.STATUSES

classmethod lru(*name*, *description=None*, *unit=''*, *default=None*, *initial_status=None*)

Instantiate a new lru sensor object.

Parameters **name** : str

The name of the sensor.

description : str

A short description of the sensor.

unit : str

The units of the sensor value. May be the empty string if there are no applicable units.

default : enum, Sensor.LRU_*

An initial value for the sensor. Defaults to self.LRU_NOMINAL

initial_status : int enum or None

An initial status for the sensor. If None, defaults to Sensor.UNKNOWN. *initial_status* must be one of the keys in Sensor.STATUSES

notify(*reading*)
Notify all observers of changes to this sensor.

classmethod parse_params(*sensor_type*, *formatted_params*, *major=5*)
Parse KATCP formatted parameters into Python values.

Parameters **sensor_type** : Sensor type constant

The type of sensor the parameters are for.

formatted_params : list of byte strings

The formatted parameters that should be parsed.

major : int

Major version of KATCP to use when interpreting types. Defaults to latest implemented KATCP version.

Returns **params** : list of objects

The parsed parameters.

classmethod parse_type(*type_string*)
Parse KATCP formatted type code into Sensor type constant.

Parameters **type_string** : str

KATCP formatted type code.

Returns **sensor_type** : Sensor type constant

The corresponding Sensor type constant.

parse_value(*s_value*, *katcp_major=5*)
Parse a value from a byte string.

Parameters **s_value** : bytes

A byte string value to attempt to convert to a value for the sensor.

Returns **value** : object

A value of a type appropriate to the sensor.

read()
Read the sensor and return a (timestamp, status, value) tuple.

Returns **reading** : `Reading` object

Sensor reading as a (timestamp, status, value) tuple.

read_formatted(*major=5*)
Read the sensor and return a (timestamp, status, value) tuple.

All values are byte strings formatted as specified in the Sensor Type Formats in the katcp specification.

Parameters **major** : int

Major version of KATCP to use when interpreting types. Defaults to latest implemented KATCP version.

Returns **timestamp** : bytes

KATCP formatted timestamp byte string

status : bytes

KATCP formatted sensor status byte string

value : bytes

KATCP formatted sensor value byte string

set(*timestamp*, *status*, *value*)
Set the current value of the sensor.

Parameters **timestamp** : float in seconds

The time at which the sensor value was determined.

status : Sensor status constant

Whether the value represents an error condition or not.

value : object

The value of the sensor (the type should be appropriate to the sensor's type).

set_formatted(*raw_timestamp*, *raw_status*, *raw_value*, *major=5*)
Set the current value of the sensor.

Parameters **raw_timestamp** : bytes

KATCP formatted timestamp byte string

raw_status : bytes

KATCP formatted sensor status byte string

raw_value : bytes

KATCP formatted sensor value byte string

major : int, default = 5

KATCP major version to use for interpreting the raw values

set_value(*value*, *status=1*, *timestamp=None*, *major=5*)
Check and then set the value of the sensor.

Parameters **value** : object

Value of the appropriate type for the sensor.

status : Sensor status constant

Whether the value represents an error condition or not.

timestamp : float in seconds or None

The time at which the sensor value was determined. Uses current time if None.

major : int

Major version of KATCP to use when interpreting types. Defaults to latest implemented KATCP version.

status()
Read the current sensor status.

Returns **status** : enum (int)

The status of the sensor, one of the keys in Sensor.STATUSES

classmethod string(*name*, *description=None*, *unit=''*, *default=None*, *initial_status=None*)
Instantiate a new string sensor object.

Parameters **name** : str

The name of the sensor.

description : str

A short description of the sensor.

unit : str

The units of the sensor value. May be the empty string if there are no applicable units.

default : string

An initial value for the sensor. Defaults to the empty string.

initial_status : int enum or None

An initial status for the sensor. If None, defaults to Sensor.UNKNOWN. *initial_status* must be one of the keys in Sensor.STATUSES

classmethod timestamp(*name*, *description=None*, *unit=''*, *default=None*, *initial_status=None*)
Instantiate a new timestamp sensor object.

Parameters **name** : str

The name of the sensor.

description : str

A short description of the sensor.

unit: str :

The units of the sensor value. For timestamp sensor may only be the empty string.

default : string

An initial value for the sensor in seconds since the Unix Epoch. Defaults to 0.

initial_status : int enum or None

An initial status for the sensor. If None, defaults to Sensor.UNKNOWN. *initial_status* must be one of the keys in Sensor.STATUSES

value()
Read the current sensor value.

Returns **value** : object

The value of the sensor (the type will be appropriate to the sensor's type).

Exceptions

class katcp.**FailReply**
Raised by request handlers to indicate a failure.

A custom exception which, when thrown in a request handler, causes DeviceServerBase to send a fail reply with the specified fail message, bypassing the generic exception handling, which would send a fail reply with a full traceback.

Examples

```
>>> class MyDevice(DeviceServer):
...     def request_myreq(self, req, msg):
...         raise FailReply("This request always fails.")
...
```

class katcp.**AsyncReply**

Raised by a request handlers to indicate it will reply later.

A custom exception which, when thrown in a request handler, indicates to DeviceServerBase that no reply has been returned by the handler but that the handler has arranged for a reply message to be sent at a later time.

Examples

```
>>> class MyDevice(DeviceServer):
...     def request_myreq(self, req, msg):
...         self.callback_client.request(
...             Message.request("otherreq"),
...             reply_cb=self._send_reply,
...         )
...         raise AsyncReply()
...
```

class katcp.**KatcpDeviceError**

Raised by KATCP servers when errors occur.

Changed in version 0.1: Deprecated in 0.1. Servers should not raise errors if communication with a client fails – errors are simply logged instead.

1.2.3 High Level Clients

KATCPClientResource

class katcp.**KATCPClientResource**(*resource_spec*, *parent=None*, *logger=<logging.Logger object>*)

Class managing a client connection to a single KATCP resource

Inspects the KATCP interface of the resources, exposing sensors and requests as per the *katcp.resource.KATCPResource* API. Can also operate without exposing

Methods

KATCPClientResource.drop_sampling_strategy(...)	Drop the sampling strategy for the named sensor from the cache
KATCPClientResource.inspecting_client_factory(...)	Return an instance of ReplyWrappedInspectingClientAsync or similar
KATCPClientResource.is_active()	
KATCPClientResource.is_connected()	Indication of the connection state
KATCPClientResource.list_sensors([filter, ...])	List sensors available on this resource matching certain criteria.

Continued on next page

Table 10 – continued from previous page

KATCPClientResource.next()	
KATCPClientResource.set_active(active)	
KATCPClientResource.set_ioloop([ioloop])	Set the tornado ioloop to use
KATCPClientResource.set_sampling_strategies(...)	Set a strategy for all sensors matching the filter, including unseen sensors The strategy should persist across sensor disconnect/reconnect.
KATCPClientResource.set_sampling_strategy(...)	Set a strategy for a sensor even if it is not yet known.
KATCPClientResource.set_sensor_listener(**kwargs)	Set a sensor listener for a sensor even if it is not yet known The listener registration should persist across sensor disconnect/reconnect.
KATCPClientResource.start()	Start the client and connect
KATCPClientResource.stop()	
KATCPClientResource.until_not_synced([timeout])	Convenience method to wait (with Future) until client is not synced
KATCPClientResource.until_state(state[, timeout])	Future that resolves when a certain client state is attained
KATCPClientResource.until_stopped([timeout])	Return future that resolves when the inspecting client has stopped
KATCPClientResource.until_synced([timeout])	Convenience method to wait (with Future) until client is synced
KATCPClientResource.wait(**kwargs)	Wait for a sensor in this resource to satisfy a condition.
KATCPClientResource.wait_connected([timeout])	Future that resolves when the state is not 'disconnected'.

MAX_LOOP_LATENCY = 0.03

When doing potentially tight loops in coroutines yield tornado.gen.moment after this much time. This is a suggestion for methods to use.

drop_sampling_strategy(*sensor_name*)

Drop the sampling strategy for the named sensor from the cache

Calling *set_sampling_strategy()* requires the requested strategy to be memorised so that it can automatically be reapplied. This method causes the strategy to be forgotten. There is no change to the current strategy. No error is raised if there is no strategy to drop.

Parameters **sensor_name** : str

Name of the sensor

inspecting_client_factory(*host*, *port*, *ioloop_set_to*)

Return an instance of ReplyWrappedInspectingClientAsync or similar

Provided to ease testing. Dynamically overriding this method after instantiation but before start() is called allows for deep brain surgery. See katcp.fake_clients.fake_inspecting_client_factory

is_connected()

Indication of the connection state

Returns True if state is not "disconnected", i.e "syncing" or "synced"

list_sensors(*filter=''*, *strategy=False*, *status=''*, *use_python_identifiers=True*, *tuple=False*, *refresh=False*)

List sensors available on this resource matching certain criteria.

Parameters **filter** : string, optional

Filter each returned sensor's name against this regexp if specified. To ease the dichotomy between Python identifier names and actual sensor names, the default is to search on Python identifier names rather than KATCP sensor names, unless *use_python_identifiers* below is set to False. Note that the sensors of subordinate KATCPResource instances may have inconsistent names and Python identifiers, better to always search on Python identifiers in this case.

strategy : {False, True}, optional

Only list sensors with a set strategy if True

status : string, optional

Filter each returned sensor's status against this regexp if given

use_python_identifiers : {True, False}, optional

Match on python identifiers even the the KATCP name is available.

tuple : {True, False}, optional, Default: False

Return backwards compatible tuple instead of SensorResultTuples

refresh : {True, False}, optional, Default: False

If set the sensor values will be refreshed with get_value before returning the results.

Returns **sensors** : list of SensorResultTuples, or list of tuples

List of matching sensors presented as named tuples. The *object* field is the `KATCPSensor` object associated with the sensor. Note that the name of the object may not match *name* if it originates from a subordinate device.

set_ioloop(*ioloop=None*)

Set the tornado ioloop to use

Defaults to tornado.ioloop.IOLoop.current() if set_ioloop() is not called or if ioloop=None. Must be called before start()

set_sampling_strategies(***kwargs*)

Set a strategy for all sensors matching the filter, including unseen sensors The strategy should persist across sensor disconnect/reconnect.

filter [str] Filter for sensor names

strategy_and_params [seq of str or str] As tuple contains (<strat_name>, [<strat_parm1>, ...]) where the strategy names and parameters are as defined by the KATCP spec. As str contains the same elements in space-separated form.

Returns **done** : tornado Future

Resolves when done

set_sampling_strategy(***kwargs*)

Set a strategy for a sensor even if it is not yet known. The strategy should persist across sensor disconnect/reconnect.

sensor_name [str] Name of the sensor

strategy_and_params [seq of str or str] As tuple contains (<strat_name>, [<strat_parm1>, ...]) where the strategy names and parameters are as defined by the KATCP spec. As str contains the same elements in space-separated form.

Returns **done** : tornado Future

Resolves when done

set_sensor_listener(***kwargs*)

Set a sensor listener for a sensor even if it is not yet known The listener registration should persist across sensor disconnect/reconnect.

sensor_name [str] Name of the sensor

listener [callable] Listening callable that will be registered on the named sensor when it becomes available. Callable as for `KATCPSensor.register_listener()`

start()

Start the client and connect

until_not_synced(*timeout=None*)

Convenience method to wait (with Future) until client is not synced

until_state(*state*, *timeout=None*)

Future that resolves when a certain client state is attained

Parameters **state** : str

Desired state, one of ("disconnected", "syncing", "synced")

timeout: float :

Timeout for operation in seconds.

until_stopped(*timeout=None*)

Return future that resolves when the inspecting client has stopped

See the *DeviceClient.until_stopped* docstring for parameter definitions and more info.

until_synced(*timeout=None*)

Convenience method to wait (with Future) until client is synced

wait(***kwargs*)

Wait for a sensor in this resource to satisfy a condition.

Parameters **sensor_name** : string

The name of the sensor to check

condition_or_value : obj or callable, or seq of objs or callables

If obj, sensor.value is compared with obj. If callable, condition_or_value(reading) is called, and must return True if its condition is satisfied. Since the reading is passed in, the value, status, timestamp or received_timestamp attributes can all be used in the check.

timeout : float or None

The timeout in seconds (None means wait forever)

Returns **This command returns a tornado Future that resolves with True when the** :

sensor value satisfies the condition, or False if the condition is :

still not satisfied after a given timeout period. :

Raises :class:'KATCPSensorError' :

If the sensor does not have a strategy set, or if the named sensor is not present

wait_connected(*timeout=None*)
Future that resolves when the state is not 'disconnected'.

KATCPClientResourceContainer

class katcp.**KATCPClientResourceContainer**(*resources_spec*, *logger=<logging.Logger object>*)
Class for containing multiple *KATCPClientResource* instances.

Provides aggregate *sensor* and *req* attributes containing the union of all the sensors in requests in the contained resources. Names are prefixed with <resname>_, where <resname> is the name of the resource to which the sensor / request belongs except for aggregate sensors that starts with `agg_`.

Methods

KATCPClientResourceContainer.add_child_resource_client(...)	Add a resource client to the container and start the resource connection
KATCPClientResourceContainer.add_group(...)	Add a new `ClientGroup` to container groups member.
KATCPClientResourceContainer.client_resource_factory(...)	Return an instance of *KATCPClientResource* or similar
KATCPClientResourceContainer.is_active()	
KATCPClientResourceContainer.is_connected()	Indication of the connection state of all children
KATCPClientResourceContainer.list_sensors([...])	List sensors available on this resource matching certain criteria.
KATCPClientResourceContainer.next()	
KATCPClientResourceContainer.set_active(active)	
KATCPClientResourceContainer.set_ioloop([ioloop])	Set the tornado ioloop to use
KATCPClientResourceContainer.set_sampling_strategies(...)	Set sampling strategies for filtered sensors - these sensors have to exists.
KATCPClientResourceContainer.set_sampling_strategy(...)	Set sampling strategies for the specific sensor - this sensor has to exist
KATCPClientResourceContainer.set_sensor_listener(...)	Set listener for the specific sensor - this sensor has to exists.
KATCPClientResourceContainer.start()	Start and connect all the subordinate clients
KATCPClientResourceContainer.stop()	Stop all child resources
KATCPClientResourceContainer.until_all_children_in_state(...)	Return a tornado Future; resolves when all clients are in specified state
KATCPClientResourceContainer.until_any_child_in_state(state)	Return a tornado Future; resolves when any client is in specified state
KATCPClientResourceContainer.until_not_synced(...)	Return a tornado Future; resolves when any subordinate client is not synced

Continued on next page

Table 11 – continued from previous page

KATCPClientResourceContainer.until_stopped([...])	Return dict of futures that resolve when each child resource has stopped
KATCPClientResourceContainer.until_synced(...)	Return a tornado Future; resolves when all subordinate clients are synced
KATCPClientResourceContainer.wait(... [, timeout])	Wait for a sensor in this resource to satisfy a condition.

add_child_resource_client(*res_name*, *res_spec*)
Add a resource client to the container and start the resource connection

add_group(*group_name*, *group_client_names*)
Add a new `ClientGroup` to container groups member.

Add the group named *group_name* with sequence of client names to the container groups member. From there it will be wrapped appropriately in the higher-level thread-safe container.

client_resource_factory(*res_spec*, *parent*, *logger*)
Return an instance of *KATCPClientResource* or similar

Provided to ease testing. Overriding this method allows deep brain surgery. See `katcp.fake_clients.fake_KATCP_client_resource_factory()`

is_connected()
Indication of the connection state of all children

list_sensors(*filter=''*, *strategy=False*, *status=''*, *use_python_identifiers=True*, *tuple=False*, *refresh=False*)
List sensors available on this resource matching certain criteria.

Parameters **filter** : string, optional

Filter each returned sensor's name against this regexp if specified. To ease the dichotomy between Python identifier names and actual sensor names, the default is to search on Python identifier names rather than KATCP sensor names, unless *use_python_identifiers* below is set to False. Note that the sensors of subordinate KATCPResource instances may have inconsistent names and Python identifiers, better to always search on Python identifiers in this case.

strategy : {False, True}, optional

Only list sensors with a set strategy if True

status : string, optional

Filter each returned sensor's status against this regexp if given

use_python_identifiers : {True, False}, optional

Match on python identifiers even the the KATCP name is available.

tuple : {True, False}, optional, Default: False

Return backwards compatible tuple instead of SensorResultTuples

refresh : {True, False}, optional, Default: False

If set the sensor values will be refreshed with get_value before returning the results.

Returns **sensors** : list of SensorResultTuples, or list of tuples

List of matching sensors presented as named tuples. The *object* field is the `KATCPSensor` object associated with the sensor. Note that the name of the object may not match *name* if it originates from a subordinate device.

set_ioloop(*ioloop=None*)
Set the tornado ioloop to use

Defaults to tornado.ioloop.IOLoop.current() if set_ioloop() is not called or if ioloop=None. Must be called before start()

set_sampling_strategies(***kwargs*)
Set sampling strategies for filtered sensors - these sensors have to exists.

set_sampling_strategy(***kwargs*)
Set sampling strategies for the specific sensor - this sensor has to exist

set_sensor_listener(***kwargs*)
Set listener for the specific sensor - this sensor has to exists.

start()
Start and connect all the subordinate clients

stop()
Stop all child resources

until_all_children_in_state(***kwargs*)
Return a tornado Future; resolves when all clients are in specified state

until_any_child_in_state(*state*, *timeout=None*)
Return a tornado Future; resolves when any client is in specified state

until_not_synced(***kwargs*)
Return a tornado Future; resolves when any subordinate client is not synced

until_stopped(*timeout=None*)
Return dict of futures that resolve when each child resource has stopped

See the *DeviceClient.until_stopped* docstring for parameter definitions and more info.

until_synced(***kwargs*)
Return a tornado Future; resolves when all subordinate clients are synced

wait(*sensor_name*, *condition_or_value*, *timeout=5*)
Wait for a sensor in this resource to satisfy a condition.

Parameters **sensor_name** : string

The name of the sensor to check

condition_or_value : obj or callable, or seq of objs or callables

If obj, sensor.value is compared with obj. If callable, condition_or_value(reading) is called, and must return True if its condition is satisfied. Since the reading is passed in, the value, status, timestamp or received_timestamp attributes can all be used in the check.

timeout : float or None

The timeout in seconds (None means wait forever)

Returns **This command returns a tornado Future that resolves with True when the** :

sensor value satisfies the condition, or False if the condition is :

still not satisfied after a given timeout period. :

Raises **:class:‘KATCPSensorError‘** :

If the sensor does not have a strategy set, or if the named sensor is not present

1.2.4 Message Parsing

Message

class katcp.**Message**(*mtype*, *name*, *arguments=None*, *mid=None*)

Represents a KAT device control language message.

Parameters **mtype** : Message type constant

The message type (request, reply or inform).

name : str

The message name.

arguments : list of objects (float, int, bool, bytes, or str)

The message arguments.

mid : str or bytes (digits only), int, or None

The message identifier. Replies and informs that are part of the reply to a request should have the same id as the request did.

Methods

Message.copy()	Return a shallow copy of the message object and its arguments.
Message.format_argument(arg)	Format a Message argument to a byte string
Message.inform(name, *args, **kwargs)	Helper method for creating inform messages.
Message.reply(name, *args, **kwargs)	Helper method for creating reply messages.
Message.reply_inform(req_msg, *args)	Helper method for creating inform messages in reply to a request.
Message.reply_ok()	Return True if this is a reply and its first argument is 'ok'.
Message.reply_to_request(req_msg, *args)	Helper method for creating reply messages to a specific request.
Message.request(name, *args, **kwargs)	Helper method for creating request messages.

copy()

Return a shallow copy of the message object and its arguments.

Returns **msg** : Message

A copy of the message object.

format_argument(*arg*)

Format a Message argument to a byte string

classmethod inform(*name*, **args*, ***kwargs*)

Helper method for creating inform messages.

Parameters **name** : str

The name of the message.

args : list of objects (float, int, bool, bytes, or str)

The message arguments.

Keyword Arguments **mid** : str or bytes (digits only), int, or None

Message ID to use or None (default) for no Message ID

classmethod reply(*name*, **args*, ***kwargs*)

Helper method for creating reply messages.

Parameters **name** : str

The name of the message.

args : list of objects (float, int, bool, bytes, or str)

The message arguments.

Keyword Arguments **mid** : str or bytes (digits only), int, or None

Message ID to use or None (default) for no Message ID

classmethod reply_inform(*req_msg*, **args*)

Helper method for creating inform messages in reply to a request.

Copies the message name and message identifier from request message.

Parameters **req_msg** : katcp.core.Message instance

The request message that this inform if in reply to

args : list of objects (float, int, bool, bytes, or str)

The message arguments except name

reply_ok()

Return True if this is a reply and its first argument is 'ok'.

classmethod reply_to_request(*req_msg*, **args*)

Helper method for creating reply messages to a specific request.

Copies the message name and message identifier from request message.

Parameters **req_msg** : katcp.core.Message instance

The request message that this inform if in reply to

args : list of objects (float, int, bool, bytes, or str)

The message arguments.

classmethod request(*name*, **args*, ***kwargs*)

Helper method for creating request messages.

Parameters **name** : str

The name of the message.

args : list of objects (float, int, bool, bytes, or str)

The message arguments.

Keyword Arguments **mid** : str or bytes (digits only), int, or None

Message ID to use or None (default) for no Message ID

MessageParser

class katcp.**MessageParser**

Parses lines into Message objects.

Methods

MessageParser.parse(line)	Parse a line, return a Message.

parse (*line*)
Parse a line, return a Message.

Parameters **line** : bytes

The line to parse (should not contain the terminating newline or carriage return).

Returns **msg** : Message object

The resulting Message.

Exceptions

class katcp.**KatcpSyntaxError**
Raised by parsers when encountering a syntax error.

1.2.5 Other

DeviceMetaclass

class katcp.**DeviceMetaclass** (*name*, *bases*, *dct*)
Metaclass for DeviceServer and DeviceClient classes.

Collects up methods named request_* and adds them to a dictionary of supported methods on the class. All request_* methods must have a doc string so that help can be generated. The same is done for inform_* and reply_* methods.

Methods

DeviceMetaclass.check_protocol(handler)	Return False if *handler* should be filtered
DeviceMetaclass.mro()	return a type's method resolution order

check_protocol (*handler*)
Return False if *handler* should be filtered

1.2.6 Version Information

katcp.**VERSION**

Five-element tuple containing the version number.

katcp.**VERSION_STR**

String representing the version number.

1.3 Kattypes

Utilities for dealing with KATCP types.

class katcp.kattypes.**Address**(*default=None*, *optional=False*, *multiple=False*)
Bases: *katcp.kattypes.KatcpType*

The KATCP address type.

Note: The address type was added in katcp 0.4.

Methods

Address.check(value, major)	Check whether the value is valid.
Address.decode(value, major)	
Address.encode(value, major)	
Address.get_default()	Return the default value.
Address.next()	
Address.pack(value[, nocheck, major])	Return the value formatted as a KATCP parameter.
Address.unpack(packed_value[, major])	Parse a KATCP parameter into an object.

class katcp.kattypes.**Bool**(*default=None*, *optional=False*, *multiple=False*)
Bases: *katcp.kattypes.KatcpType*

The KATCP boolean type.

Methods

Bool.check(value, major)	Check whether the value is valid.
Bool.decode(value, major)	
Bool.encode(value, major)	
Bool.get_default()	Return the default value.
Bool.next()	
Bool.pack(value[, nocheck, major])	Return the value formatted as a KATCP parameter.
Bool.unpack(packed_value[, major])	Parse a KATCP parameter into an object.

class katcp.kattypes.**Discrete**(*values*, *case_insensitive=False*, ***kwargs*)
Bases: *katcp.kattypes.Str*

The KATCP discrete type.

Parameters **values** : iterable of str

List of the values the discrete type may accept.

case_insensitive : bool

Whether case-insensitive value matching should be used.

Methods

Discrete.check(value, major)	Check whether the value in the set of allowed values.
Discrete.decode(value, major)	
Discrete.encode(value, major)	
Discrete.get_default()	Return the default value.
Discrete.next()	
Discrete.pack(value[, nocheck, major])	Return the value formatted as a KATCP parameter.
Discrete.unpack(packed_value[, major])	Parse a KATCP parameter into an object.

check(*value*, *major*)

Check whether the value in the set of allowed values.

Raise a ValueError if it is not.

class katcp.kattypes.**DiscreteMulti**(*values*, *all_keyword='all'*, *separator=','*, ***kwargs*)

Bases: *katcp.kattypes.Discrete*

Discrete type which can accept multiple values.

Its value is always a list.

Parameters **values** : list of str

Set of allowed values.

all_keyword : str, optional

The string which represents the list of all allowed values.

separator : str, optional

The separator used in the packed value string.

Methods

DiscreteMulti.check(value, major)	Check that each item in the value list is in the allowed set.
DiscreteMulti.decode(value, major)	
DiscreteMulti.encode(value, major)	
DiscreteMulti.get_default()	Return the default value.
DiscreteMulti.next()	
DiscreteMulti.pack(value[, nocheck, major])	Return the value formatted as a KATCP parameter.
DiscreteMulti.unpack(packed_value[, major])	Parse a KATCP parameter into an object.

check(*value*, *major*)

Check that each item in the value list is in the allowed set.

class katcp.kattypes.**Float**(*min=None*, *max=None*, ***kwargs*)

Bases: *katcp.kattypes.KatcpType*

The KATCP float type.

Parameters **min** : float

The minimum allowed value. Ignored if not given.

max : float

The maximum allowed value. Ignored if not given.

Methods

Float.check(value, major)	Check whether the value is between the minimum and maximum.
Float.decode(value, major)	
Float.encode(value, major)	
Float.get_default()	Return the default value.
Float.next()	
Float.pack(value[, nocheck, major])	Return the value formatted as a KATCP parameter.
Float.unpack(packed_value[, major])	Parse a KATCP parameter into an object.

check(*value*, *major*)

Check whether the value is between the minimum and maximum.

Raise a ValueError if it is not.

class katcp.kattypes.**Int**(*min=None*, *max=None*, ***kwargs*)

Bases: *katcp.kattypes.KatcpType*

The KATCP integer type.

Parameters **min** : int

The minimum allowed value. Ignored if not given.

max : int

The maximum allowed value. Ignored if not given.

Methods

Int.check(value, major)	Check whether the value is between the minimum and maximum.
Int.decode(value, major)	
Int.encode(value, major)	
Int.get_default()	Return the default value.
Int.next()	
Int.pack(value[, nocheck, major])	Return the value formatted as a KATCP parameter.
Int.unpack(packed_value[, major])	Parse a KATCP parameter into an object.

check(*value*, *major*)

Check whether the value is between the minimum and maximum.

Raise a ValueError if it is not.

class katcp.kattypes.**KatcpType**(*default=None*, *optional=False*, *multiple=False*)

Bases: future.types.newobject.newobject

Class representing a KATCP type.

Sub-classes should:

- Set the name attribute.

- Implement the `encode()` method.
- Implement the `decode()` method.

Parameters **default** : object, optional

The default value for this type.

optional : boolean, optional

Whether the value is allowed to be None.

multiple : boolean, optional

Whether multiple values of this type are expected. Must be the last type parameter if this is True.

Methods

KatcpType.check(value, major)	Check whether the value is valid.
KatcpType.get_default()	Return the default value.
KatcpType.next()	
KatcpType.pack(value[, nocheck, major])	Return the value formatted as a KATCP parameter.
KatcpType.unpack(packed_value[, major])	Parse a KATCP parameter into an object.

check (*value*, *major*)
Check whether the value is valid.

Do nothing if the value is valid. Raise an exception if the value is not valid. Parameter major describes the KATCP major version to use when interpreting the validity of a value.

get_default ()
Return the default value.

Raise a ValueError if the value is not optional and there is no default.

Returns **default** : object

The default value.

pack (*value*, *nocheck=False*, *major=5*)
Return the value formatted as a KATCP parameter.

Parameters **value** : object

The value to pack.

nocheck : bool, optional

Whether to check that the value is valid before packing it.

major : int, optional

Major version of KATCP to use when interpreting types. Defaults to latest implemented KATCP version.

Returns **packed_value** : bytes

The unescaped KATCP byte string representing the value.

unpack (*packed_value*, *major=5*)
Parse a KATCP parameter into an object.

Parameters **packed_value** : bytes

The unescaped KATCP byte string to parse into a value.

major : int, optional

Major version of KATCP to use when interpreting types. Defaults to latest implemented KATCP version.

Returns **value** : object

The value the KATCP string represented.

class katcp.kattypes.**Lru**(*default=None*, *optional=False*, *multiple=False*)

Bases: *katcp.kattypes.KatcpType*

The KATCP lru type

Methods

Lru.check(value, major)	Check whether the value is valid.
Lru.decode(value, major)	
Lru.encode(value, major)	
Lru.get_default()	Return the default value.
Lru.next()	
Lru.pack(value[, nocheck, major])	Return the value formatted as a KATCP parameter.
Lru.unpack(packed_value[, major])	Parse a KATCP parameter into an object.

class katcp.kattypes.**Parameter**(*position*, *name*, *kattype*, *major*)

Bases: future.types.newobject.newobject

Wrapper for kattypes which holds parameter-specific information.

Parameters **position** : int

The parameter's position (starts at 1)

name : str

The parameter's name (introspected)

kattype : KatcpType object

The parameter's kattype

major : integer

Major version of KATCP to use when interpreting types

Methods

Parameter.next()	
Parameter.pack(value)	Pack the parameter using its kattype.
Parameter.unpack(value)	Unpack the parameter using its kattype.

pack(*value*)

Pack the parameter using its kattype.

Parameters **value** : object

The value to pack

Returns **packed_value** : str

The unescaped KATCP string representing the value.

unpack(*value*)
Unpack the parameter using its kattype.

Parameters **packed_value** : str

The unescaped KATCP string to unpack.

Returns **value** : object

The unpacked value.

class katcp.kattypes.**Regex**(*regex*, ***kwargs*)
Bases: *katcp.kattypes.Str*

String type that checks values using a regular expression.

Parameters **regex** : str or regular expression object

Regular expression that values should match.

Methods

Regex.check(value, major)	Check whether the value is valid.
Regex.decode(value, major)	
Regex.encode(value, major)	
Regex.get_default()	Return the default value.
Regex.next()	
Regex.pack(value[, nocheck, major])	Return the value formatted as a KATCP parameter.
Regex.unpack(packed_value[, major])	Parse a KATCP parameter into an object.

check(*value*, *major*)
Check whether the value is valid.

Do nothing if the value is valid. Raise an exception if the value is not valid. Parameter major describes the KATCP major version to use when interpreting the validity of a value.

class katcp.kattypes.**Str**(*default=None*, *optional=False*, *multiple=False*)
Bases: *katcp.kattypes.KatcpType*

The KATCP string type.

Notes

The behaviour of this type is subtly different between Python versions in order to ease the porting effort for users of this library. - Unpacked (decoded) values are native strings (bytes in PY2, Unicode in PY3). - Packed (encoded) values are always byte strings (in both PY2 and PY3), as this is what is sent on the wire.

UTF-8 encoding is used when converting between Unicode and byte strings. Thus ASCII values are fine, but arbitrary strings of bytes are not safe to use, and may raise an exception.

For convenience, non-text types can be encoded. The object is converted to a string, and then to bytes. This is a one-way operation - when that byte string is decoded the original type will not be recovered.

Methods

`Str.check`(value, major)	Check whether the value is valid.
`Str.decode`(value, major)	
`Str.encode`(value, major)	
`Str.get_default`()	Return the default value.
`Str.next`()	
`Str.pack`(value[, nocheck, major])	Return the value formatted as a KATCP parameter.
`Str.unpack`(packed_value[, major])	Parse a KATCP parameter into an object.

class katcp.kattypes.**StrictTimestamp**(*default=None*, *optional=False*, *multiple=False*)

Bases: *katcp.kattypes.KatcpType*

A timestamp that enforces the XXXX.YYY format for timestamps.

Methods

StrictTimestamp.check(value, major)	Check whether the value is positive.
`StrictTimestamp.decode`(value, major)	
`StrictTimestamp.encode`(value, major)	
`StrictTimestamp.get_default`()	Return the default value.
`StrictTimestamp.next`()	
`StrictTimestamp.pack`(value[, nocheck, major])	Return the value formatted as a KATCP parameter.
`StrictTimestamp.unpack`(packed_value[, major])	Parse a KATCP parameter into an object.

check(*value*, *major*)

Check whether the value is positive.

Raise a ValueError if it is not.

class katcp.kattypes.**Struct**(*fmt*, ***kwargs*)

Bases: *katcp.kattypes.KatcpType*

KatcpType for parsing and packing values using the `struct` module.

Parameters **fmt** : str

Format to use for packing and unpacking values. It is passed directly into `struct.pack()` and `struct.unpack()`.

Methods

`Struct.check`(value, major)	Check whether the value is valid.
`Struct.decode`(value, major)	
`Struct.encode`(value, major)	
`Struct.get_default`()	Return the default value.
`Struct.next`()	
`Struct.pack`(value[, nocheck, major])	Return the value formatted as a KATCP parameter.
`Struct.unpack`(packed_value[, major])	Parse a KATCP parameter into an object.

class katcp.kattypes.**Timestamp**(*default=None*, *optional=False*, *multiple=False*)
Bases: *katcp.kattypes.KatcpType*

The KATCP timestamp type.

Methods

Timestamp.check(value, major)	Check whether the value is valid.
Timestamp.decode(value, major)	
Timestamp.encode(value, major)	
Timestamp.get_default()	Return the default value.
Timestamp.next()	
Timestamp.pack(value[, nocheck, major])	Return the value formatted as a KATCP parameter.
Timestamp.unpack(packed_value[, major])	Parse a KATCP parameter into an object.

class katcp.kattypes.**TimestampOrNow**(*default=None*, *optional=False*, *multiple=False*)
Bases: *katcp.kattypes.Timestamp*

KatcpType representing either a Timestamp or the special value for now.

Floats are encoded as for *katcp.kattypes.Timestamp*. The special value for now, katcp.kattypes.TimestampOrNow.NOW, is encoded as the string "now".

Methods

TimestampOrNow.check(value, major)	Check whether the value is valid.
TimestampOrNow.decode(value, major)	
TimestampOrNow.encode(value, major)	
TimestampOrNow.get_default()	Return the default value.
TimestampOrNow.next()	
TimestampOrNow.pack(value[, nocheck, major])	Return the value formatted as a KATCP parameter.
TimestampOrNow.unpack(packed_value[, major])	Parse a KATCP parameter into an object.

katcp.kattypes.**async_make_reply**(**args*, ***kwargs*)
Wrap future that will resolve with arguments needed by make_reply().

katcp.kattypes.**concurrent_reply**(*handler*)
Decorator for concurrent async request handlers

By default async request handlers that return a Future are serialised per-connection, i.e. until the most recent handler resolves its future, the next message will not be read from the client stream. A handler decorated with this decorator allows the next message to be read before it has resolved its future, allowing multiple requests from a single client to be handled concurrently. This is similar to raising AsyncReply.

Examples

```
>>> class MyDevice(DeviceServer):
...     @return_reply(Int())
...     @concurrent_reply
```

(continues on next page)

(continued from previous page)

```
...     @tornado.gen.coroutine
...     def request_myreq(self, req):
...         '''A slow request'''
...         result = yield self.slow_operation()
...         raise tornado.gen.Return((req, result))
...
```

katcp.kattypes.**has_katcp_protocol_flags**(*protocol_flags*)

Decorator; only include handler if server has these protocol flags

Useful for including default handler implementations for KATCP features that are only present when certain server protocol flags are set.

Examples

```
>>> class MyDevice(DeviceServer):
...     '''This device server will expose ?myreq'''
...     PROTOCOL_INFO = katcp.core.ProtocolFlags(5, 0, [
                    katcp.core.ProtocolFlags.MULTI_CLIENT])
...
...     @has_katcp_protocol_flags([katcp.core.ProtocolFlags.MULTI_CLIENT])
...     def request_myreq(self, req, msg):
...         '''A request that requires multi-client support'''
...         # Request handler implementation here.
...
>>> class MySingleClientDevice(MyDevice):
...     '''This device server will not expose ?myreq'''
...
...     PROTOCOL_INFO = katcp.core.ProtocolFlags(5, 0, [])
...
```

katcp.kattypes.**inform**()

Decorator for inform handler methods.

The method being decorated should take arguments matching the list of types. The decorator will unpack the request message into the arguments.

Parameters **types** : list of kattypes

The types of the request message parameters (in order). A type with multiple=True has to be the last type.

Keyword Arguments **include_msg** : bool, optional

Pass the request message as the third parameter to the decorated request handler function (default is False).

major : int, optional

Major version of KATCP to use when interpreting types. Defaults to latest implemented KATCP version.

Examples

```
>>> class MyDeviceClient(katcp.client.AsyncClient):
...     @inform(Int(), Float())
...     def inform_myinf(self, my_int, my_float):
...         '''Handle #myinf <my_int> <my_float> inform received from server'''
...         # Call some code here that reacts to my_inf and my_float
```

katcp.kattypes.**make_reply**(*msgname*, *types*, *arguments*, *major*)

Helper method for constructing a reply message from a list or tuple.

Parameters **msgname** : str

Name of the reply message.

types : list of kattypes

The types of the reply message parameters (in order).

arguments : list of objects

The (unpacked) reply message parameters.

major : integer

Major version of KATCP to use when packing types

katcp.kattypes.**minimum_katcp_version**(*major*, *minor=0*)

Decorator; exclude handler if server's protocol version is too low

Useful for including default handler implementations for KATCP features that are only present in certain KATCP protocol versions

Examples

```
>>> class MyDevice(DeviceServer):
...     '''This device server will expose ?myreq'''
...     PROTOCOL_INFO = katcp.core.ProtocolFlags(5, 1)
...
...     @minimum_katcp_version(5, 1)
...     def request_myreq(self, req, msg):
...         '''A request that should only be present for KATCP >v5.1'''
...         # Request handler implementation here.
...
>>> class MyOldDevice(MyDevice):
...     '''This device server will not expose ?myreq'''
...
...     PROTOCOL_INFO = katcp.core.ProtocolFlags(5, 0)
...
```

katcp.kattypes.**pack_types**(*types*, *args*, *major*)

Pack arguments according the the types list.

Parameters **types** : sequence of kattypes

The types of the arguments (in order).

args : sequence of objects

The arguments to format.

major : integer

Major version of KATCP to use when packing types

Returns **packed_args** : list

List of args after packing to byte strings

katcp.kattypes.**request**(**types*, ***options*)

Decorator for request handler methods.

The method being decorated should take a req argument followed by arguments matching the list of types. The decorator will unpack the request message into the arguments.

Parameters **types** : list of kattypes

The types of the request message parameters (in order). A type with multiple=True has to be the last type.

Keyword Arguments **include_msg** : bool, optional

Pass the request message as the third parameter to the decorated request handler function (default is False).

major : int, optional

Major version of KATCP to use when interpreting types. Defaults to latest implemented KATCP version.

Examples

```
>>> class MyDevice(DeviceServer):
...     @request(Int(), Float(), Bool())
...     @return_reply(Int(), Float())
...     def request_myreq(self, req, my_int, my_float, my_bool):
...         '''?myreq my_int my_float my_bool'''
...         return ("ok", my_int + 1, my_float / 2.0)
...
...     @request(Int(), include_msg=True)
...     @return_reply(Bool())
...     def request_is_odd(self, req, msg, my_int):
            '''?is-odd <my_int>, reply '1' if <my_int> is odd, else 0'''
...         req.inform('Checking oddity of %d' % my_int)
...         return ("ok", my_int % 2)
...
```

katcp.kattypes.**request_timeout_hint**(*timeout_hint*)

Decorator; add recommended client timeout hint to a request for request

Useful for requests that take longer than average to reply. Hint is provided to clients via ?request-timeout-hint. Note this is only exposed if the device server sets the protocol version to KATCP v5.1 or higher and enables the REQUEST_TIMEOUT_HINTS flag in its PROTOCOL_INFO class attribute

Parameters **timeout_hint** : float (seconds) or None

How long the decorated request should reasonably take to reply. No timeout hint if None, similar to never using the decorator, provided for consistency.

Examples

```
>>> class MyDevice(DeviceServer):
...     @return_reply(Int())
```

(continues on next page)

(continued from previous page)

```
...     @request_timeout_hint(15) # Set request timeout hint to 15 seconds
...     @tornado.gen.coroutine
...     def request_myreq(self, req):
...         '''A slow request'''
...         result = yield self.slow_operation()
...         raise tornado.gen.Return((req, result))
...
```

katcp.kattypes.**return_reply**(**types*, ***options*)

Decorator for returning replies from request handler methods.

The method being decorated should return an iterable of result values. If the first value is 'ok', the decorator will check the remaining values against the specified list of types (if any). If the first value is 'fail' or 'error', there must be only one remaining parameter, and it must be a string describing the failure or error In both cases, the decorator will pack the values into a reply message.

Parameters **types** : list of kattypes

The types of the reply message parameters (in order).

Keyword Arguments **major** : int, optional

Major version of KATCP to use when interpreting types. Defaults to latest implemented KATCP version.

Examples

```
>>> class MyDevice(DeviceServer):
...     @request(Int())
...     @return_reply(Int(), Float())
...     def request_myreq(self, req, my_int):
...         return ("ok", my_int + 1, my_int * 2.0)
...
```

katcp.kattypes.**send_reply**(**types*, ***options*)

Decorator for sending replies from request callback methods.

This decorator constructs a reply from a list or tuple returned from a callback method, but unlike the return_reply decorator it also sends the reply rather than returning it.

The list/tuple returned from the callback method must have req (a ClientRequestConnection instance) as its first parameter and the original message as the second. The original message is needed to determine the message name and ID.

The device with the callback method must have a reply method.

Parameters **types** : list of kattypes

The types of the reply message parameters (in order).

Keyword Arguments **major** : int, optional

Major version of KATCP to use when interpreting types. Defaults to latest implemented KATCP version.

Examples

```
>>> class MyDevice(DeviceServer):
...     @send_reply(Int(), Float())
...     def my_callback(self, req):
...         return (req, "ok", 5, 2.0)
...
```

katcp.kattypes.**unpack_message**()

Decorator that unpacks katcp.Messages to function arguments.

The method being decorated should take arguments matching the list of types. The decorator will unpack the request message into the arguments.

Parameters **types** : list of kattypes

The types of the request message parameters (in order). A type with multiple=True has to be the last type.

Keyword Arguments **include_msg** : bool, optional

Pass the request message as the third parameter to the decorated request handler function (default is False).

major : int, optional

Major version of KATCP to use when interpreting types. Defaults to latest implemented KATCP version.

Examples

```
>>> class MyClient(DeviceClient):
...     @unpack_message(Str(), Int(), Float(), Bool())
...     def reply_myreq(self, status, my_int, my_float, my_bool):
...         print 'myreq replied with ', (status, my_int, my_float, my_bool)
...
...     @unpack_message(Str(), Int(), include_msg=True)
...     def inform_fruit_picked(self, msg, fruit, no_picked):
...         print no_picked, 'of fruit ', fruit, ' picked.'
...         print 'Raw inform message: ', str(msg)
```

katcp.kattypes.**unpack_types**(*types*, *args*, *argnames*, *major*)

Parse arguments according to types list.

Parameters **types** : sequence of kattypes

The types of the arguments (in order).

args : sequence of strings

The arguments to parse.

argnames : sequence of strings

The names of the arguments.

major : integer

Major version of KATCP to use when packing types

Returns **unpacked_args** : list

List of args after unpacking to kattype objects

katcp.kattypes.**unpack_message**()

Decorator that unpacks katcp.Messages to function arguments.

The method being decorated should take arguments matching the list of types. The decorator will unpack the request message into the arguments.

Parameters **types** : list of kattypes

The types of the request message parameters (in order). A type with multiple=True has to be the last type.

Keyword Arguments **include_msg** : bool, optional

Pass the request message as the third parameter to the decorated request handler function (default is False).

major : int, optional

Major version of KATCP to use when interpreting types. Defaults to latest implemented KATCP version.

Examples

```
>>> class MyClient(DeviceClient):
...     @unpack_message(Str(), Int(), Float(), Bool())
...     def reply_myreq(self, status, my_int, my_float, my_bool):
...         print 'myreq replied with ', (status, my_int, my_float, my_bool)
...
...     @unpack_message(Str(), Int(), include_msg=True)
...     def inform_fruit_picked(self, msg, fruit, no_picked):
...         print no_picked, 'of fruit ', fruit, ' picked.'
...         print 'Raw inform message: ', str(msg)
```

1.4 Low level client API (client)

Clients for the KAT device control language.

class katcp.client.**AsyncClient**(*host*, *port*, *tb_limit=20*, *timeout=5.0*, *logger=<logging.Logger object>*, *auto_reconnect=True*)

Bases: *katcp.client.DeviceClient*

Implement async and callback-based requests on top of DeviceClient.

This client will use message IDs if the server supports them.

Parameters **host** : string

Host to connect to.

port : int

Port to connect to.

tb_limit : int, optional

Maximum number of stack frames to send in error traceback.

logger : object, optional

Python Logger object to log to. Default is a logger named 'katcp'.

auto_reconnect : bool, optional

Whether to automatically reconnect if the connection dies.

timeout : float in seconds, optional

Default number of seconds to wait before a callback callback_request times out. Can be overridden in individual calls to callback_request.

Examples

```
>>> def reply_cb(msg):
...     print "Reply:", msg
...
>>> def inform_cb(msg):
...     print "Inform:", msg
...
>>> c = AsyncClient('localhost', 10000)
>>> c.start()
>>> c.ioloop.add_callback(
...     c.callback_request,
...     katcp.Message.request('myreq'),
...     reply_cb=reply_cb,
...     inform_cb=inform_cb,
... )
...
>>> # expect reply to be printed here
>>> # stop the client once we're finished with it
>>> c.stop()
>>> c.join()
```

Methods

AsyncClient.blocking_request(msg[, timeout, ...])	Send a request message and wait for its reply.
AsyncClient.callback_request(msg[, ...])	Send a request message.
AsyncClient.convert_seconds(time_seconds)	Convert a time in seconds to the device timestamp units.
AsyncClient.disconnect()	Force client connection to close, reconnect if auto-connect set.
AsyncClient.enable_thread_safety()	Enable thread-safety features.
AsyncClient.future_request(msg[, timeout, ...])	Send a request message, with future replies.
AsyncClient.handle_inform(msg)	Handle inform messages related to any current requests.
AsyncClient.handle_message(msg)	Handle a message from the server.
AsyncClient.handle_reply(msg)	Handle a reply message related to the current request.
AsyncClient.handle_request(msg)	Dispatch a request message to the appropriate method.

Continued on next page

Table 30 – continued from previous page

AsyncClient.inform_build_state(msg)	Handle katcp v4 and below build-state inform.
AsyncClient.inform_version(msg)	Handle katcp v4 and below version inform.
AsyncClient.inform_version_connect(msg)	Process a #version-connect message.
AsyncClient.is_connected()	Check if the socket is currently connected.
AsyncClient.join([timeout])	Rejoin the client thread.
AsyncClient.next()	
AsyncClient.notify_connected(connected)	Event handler that is called whenever the connection status changes.
AsyncClient.preset_protocol_flags(protocol_flags)	Pre-set the protocol flags.
AsyncClient.request(msg[, use_mid])	Send a request message, with automatic message ID assignment.
AsyncClient.running()	Whether the client is running.
AsyncClient.send_message(msg)	Send any kind of message.
AsyncClient.send_request(msg)	Send a request message.
AsyncClient.set_ioloop([ioloop])	Set the tornado.ioloop.IOLoop instance to use.
AsyncClient.start([timeout])	Start the client in a new thread.
AsyncClient.stop(*args, **kwargs)	Stop a running client.
AsyncClient.unhandled_inform(msg)	Fallback method for inform messages without a registered handler.
AsyncClient.unhandled_reply(msg)	Fallback method for reply messages without a registered handler.
AsyncClient.unhandled_request(msg)	Fallback method for requests without a registered handler.
AsyncClient.until_connected(**kwargs)	Return future that resolves when the client is connected.
AsyncClient.until_protocol(**kwargs)	Return future that resolves after receipt of katcp protocol info.
AsyncClient.until_running([timeout])	Return future that resolves when the client is running.
AsyncClient.until_stopped([timeout])	Return future that resolves when the client has stopped.
AsyncClient.wait_connected([timeout])	Wait until the client is connected.
AsyncClient.wait_disconnected([timeout])	Wait until the client is disconnected.
AsyncClient.wait_protocol([timeout])	Wait until katcp protocol information has been received from server.
AsyncClient.wait_running([timeout])	Wait until the client is running.

blocking_request(*msg*, *timeout=None*, *use_mid=None*)

Send a request message and wait for its reply.

Parameters **msg** : Message object

The request Message to send.

timeout : float in seconds

How long to wait for a reply. The default is the the timeout set when creating the AsyncClient.

use_mid : boolean, optional

Whether to use message IDs. Default is to use message IDs if the server supports them.

Returns **reply** : Message object

The reply message received.

informs : list of Message objects

A list of the inform messages received.

callback_request(*msg*, *reply_cb=None*, *inform_cb=None*, *user_data=None*, *timeout=None*, *use_mid=None*)

Send a request message.

Parameters **msg** : Message object

The request message to send.

reply_cb : function

The reply callback with signature reply_cb(msg) or reply_cb(msg, *user_data)

inform_cb : function

The inform callback with signature inform_cb(msg) or inform_cb(msg, *user_data)

user_data : tuple

Optional user data to send to the reply and inform callbacks.

timeout : float in seconds

How long to wait for a reply. The default is the the timeout set when creating the AsyncClient.

use_mid : boolean, optional

Whether to use message IDs. Default is to use message IDs if the server supports them.

future_request(*msg*, *timeout=None*, *use_mid=None*)

Send a request message, with future replies.

Parameters **msg** : Message object

The request Message to send.

timeout : float in seconds

How long to wait for a reply. The default is the the timeout set when creating the AsyncClient.

use_mid : boolean, optional

Whether to use message IDs. Default is to use message IDs if the server supports them.

Returns **A tornado.concurrent.Future that resolves with:** :

reply : Message object

The reply message received.

informs : list of Message objects

A list of the inform messages received.

handle_inform(*msg*)

Handle inform messages related to any current requests.

Inform messages not related to the current request go up to the base class method.

Parameters **msg** : Message object

The inform message to dispatch.

handle_reply(*msg*)
Handle a reply message related to the current request.

Reply messages not related to the current request go up to the base class method.

Parameters **msg** : Message object

The reply message to dispatch.

stop(**args*, ***kwargs*)
Stop a running client.

If using a managed ioloop, this must be called from a different thread to the ioloop's. This method only returns once the client's main coroutine, *_install()*, has completed.

If using an unmanaged ioloop, this can be called from the same thread as the ioloop. The *until_stopped()* method can be used to wait on completion of the main coroutine, *_install()*.

Parameters **timeout** : float in seconds

Seconds to wait for both client thread to have *started*, and for stopping.

class katcp.client.**BlockingClient**(*host*, *port*, *tb_limit=20*, *timeout=5.0*, *logger=<logging.Logger object>*, *auto_reconnect=True*)
Bases: *katcp.client.CallbackClient*

Methods

BlockingClient.blocking_request(msg[, ...])	Send a request message and wait for its reply.
BlockingClient.callback_request(msg[, ...])	Send a request message.
BlockingClient.convert_seconds(time_seconds)	Convert a time in seconds to the device timestamp units.
BlockingClient.disconnect()	Force client connection to close, reconnect if auto-connect set.
BlockingClient.enable_thread_safety()	Enable thread-safety features.
BlockingClient.future_request(msg[, ...])	Send a request message, with future replies.
BlockingClient.handle_inform(msg)	Handle inform messages related to any current requests.
BlockingClient.handle_message(msg)	Handle a message from the server.
BlockingClient.handle_reply(msg)	Handle a reply message related to the current request.
BlockingClient.handle_request(msg)	Dispatch a request message to the appropriate method.
BlockingClient.inform_build_state(msg)	Handle katcp v4 and below build-state inform.
BlockingClient.inform_version(msg)	Handle katcp v4 and below version inform.
BlockingClient.inform_version_connect(msg)	Process a #version-connect message.
BlockingClient.is_connected()	Check if the socket is currently connected.
BlockingClient.join([timeout])	Rejoin the client thread.
BlockingClient.next()	

Continued on next page

Table 31 – continued from previous page

BlockingClient.notify_connected(connected)	Event handler that is called whenever the connection status changes.
BlockingClient.preset_protocol_flags(...)	Preset server protocol flags.
BlockingClient.request(msg[, use_mid])	Send a request message, with automatic message ID assignment.
BlockingClient.running()	Whether the client is running.
BlockingClient.send_message(msg)	Send any kind of message.
BlockingClient.send_request(msg)	Send a request message.
BlockingClient.setDaemon(daemonic)	Set daemonic state of the managed ioloop thread to True / False
BlockingClient.set_ioloop([ioloop])	Set the tornado.ioloop.IOLoop instance to use.
BlockingClient.start([timeout])	Start the client in a new thread.
BlockingClient.stop(*args, **kwargs)	Stop a running client.
BlockingClient.unhandled_inform(msg)	Fallback method for inform messages without a registered handler.
BlockingClient.unhandled_reply(msg)	Fallback method for reply messages without a registered handler.
BlockingClient.unhandled_request(msg)	Fallback method for requests without a registered handler.
BlockingClient.until_connected(**kwargs)	Return future that resolves when the client is connected.
BlockingClient.until_protocol(**kwargs)	Return future that resolves after receipt of katcp protocol info.
BlockingClient.until_running([timeout])	Return future that resolves when the client is running.
BlockingClient.until_stopped([timeout])	Return future that resolves when the client has stopped.
BlockingClient.wait_connected([timeout])	Wait until the client is connected.
BlockingClient.wait_disconnected([timeout])	Wait until the client is disconnected.
BlockingClient.wait_protocol([timeout])	Wait until katcp protocol information has been received from server.
BlockingClient.wait_running([timeout])	Wait until the client is running.

class katcp.client.**CallbackClient**(*host*, *port*, *tb_limit=20*, *timeout=5.0*, *logger=<logging.Logger object>*, *auto_reconnect=True*)

Bases: *katcp.client.AsyncClient*

Methods

CallbackClient.blocking_request(msg[, ...])	Send a request message and wait for its reply.
CallbackClient.callback_request(msg[, ...])	Send a request message.
CallbackClient.convert_seconds(time_seconds)	Convert a time in seconds to the device timestamp units.
CallbackClient.disconnect()	Force client connection to close, reconnect if auto-connect set.

Continued on next page

Table 32 – continued from previous page

CallbackClient.enable_thread_safety()	Enable thread-safety features.
CallbackClient.future_request(msg[, ...])	Send a request message, with future replies.
CallbackClient.handle_inform(msg)	Handle inform messages related to any current requests.
CallbackClient.handle_message(msg)	Handle a message from the server.
CallbackClient.handle_reply(msg)	Handle a reply message related to the current request.
CallbackClient.handle_request(msg)	Dispatch a request message to the appropriate method.
CallbackClient.inform_build_state(msg)	Handle katcp v4 and below build-state inform.
CallbackClient.inform_version(msg)	Handle katcp v4 and below version inform.
CallbackClient.inform_version_connect(msg)	Process a #version-connect message.
CallbackClient.is_connected()	Check if the socket is currently connected.
CallbackClient.join([timeout])	Rejoin the client thread.
CallbackClient.next()	
CallbackClient.notify_connected(connected)	Event handler that is called whenever the connection status changes.
CallbackClient.preset_protocol_flags(...)	Preset server protocol flags.
CallbackClient.request(msg[, use_mid])	Send a request message, with automatic message ID assignment.
CallbackClient.running()	Whether the client is running.
CallbackClient.send_message(msg)	Send any kind of message.
CallbackClient.send_request(msg)	Send a request message.
CallbackClient.setDaemon(daemonic)	Set daemonic state of the managed ioloop thread to True / False
CallbackClient.set_ioloop([ioloop])	Set the tornado.ioloop.IOLoop instance to use.
CallbackClient.start([timeout])	Start the client in a new thread.
CallbackClient.stop(*args, **kwargs)	Stop a running client.
CallbackClient.unhandled_inform(msg)	Fallback method for inform messages without a registered handler.
CallbackClient.unhandled_reply(msg)	Fallback method for reply messages without a registered handler.
CallbackClient.unhandled_request(msg)	Fallback method for requests without a registered handler.
CallbackClient.until_connected(**kwargs)	Return future that resolves when the client is connected.
CallbackClient.until_protocol(**kwargs)	Return future that resolves after receipt of katcp protocol info.
CallbackClient.until_running([timeout])	Return future that resolves when the client is running.
CallbackClient.until_stopped([timeout])	Return future that resolves when the client has stopped.
CallbackClient.wait_connected([timeout])	Wait until the client is connected.
CallbackClient.wait_disconnected([timeout])	Wait until the client is disconnected.

Continued on next page

Table 32 – continued from previous page

`CallbackClient.wait_protocol`([timeout])	Wait until katcp protocol information has been received from server.
`CallbackClient.wait_running`([timeout])	Wait until the client is running.

setDaemon(*daemonic*)

Set daemonic state of the managed ioloop thread to True / False

Calling this method for a non-managed ioloop has no effect. Must be called before start(), or it will also have no effect

class katcp.client.**DeviceClient**(*host*, *port*, *tb_limit=20*, *logger=<logging.Logger object>*, *auto_reconnect=True*)

Bases: `future.types.newobject.newobject`

Device client proxy.

Subclasses should implement .reply_*, .inform_* and send_request_* methods to take actions when messages arrive, and implement unhandled_inform, unhandled_reply and unhandled_request to provide fallbacks for messages for which there is no handler.

Request messages can be sent by calling .send_request().

Parameters **host** : string

Host to connect to.

port : int

Port to connect to.

tb_limit : int

Maximum number of stack frames to send in error traceback.

logger : object

Python Logger object to log to.

auto_reconnect : bool

Whether to automatically reconnect if the connection dies.

Notes

The client may block its ioloop if the default blocking tornado DNS resolver is used. When an ioloop is shared, it would make sense to configure one of the non-blocking resolver classes, see http://tornado.readthedocs.org/en/latest/netutil.html

Examples

```
>>> MyClient(DeviceClient):
...     def reply_myreq(self, msg):
...         print str(msg)
...
>>> c = MyClient('localhost', 10000){
>>> c.start()
>>> c.send_request(katcp.Message.request('myreq'))
>>> # expect reply to be printed here
>>> # stop the client once we're finished with it
```

(continues on next page)

(continued from previous page)

```
>>> c.stop()
>>> c.join()
```

Methods

DeviceClient.convert_seconds(time_seconds)	Convert a time in seconds to the device timestamp units.
DeviceClient.disconnect()	Force client connection to close, reconnect if auto-connect set.
DeviceClient.enable_thread_safety()	Enable thread-safety features.
DeviceClient.handle_inform(msg)	Dispatch an inform message to the appropriate method.
DeviceClient.handle_message(msg)	Handle a message from the server.
DeviceClient.handle_reply(msg)	Dispatch a reply message to the appropriate method.
DeviceClient.handle_request(msg)	Dispatch a request message to the appropriate method.
DeviceClient.inform_build_state(msg)	Handle katcp v4 and below build-state inform.
DeviceClient.inform_version(msg)	Handle katcp v4 and below version inform.
DeviceClient.inform_version_connect(msg)	Process a #version-connect message.
DeviceClient.is_connected()	Check if the socket is currently connected.
DeviceClient.join([timeout])	Rejoin the client thread.
DeviceClient.next()	
DeviceClient.notify_connected(connected)	Event handler that is called whenever the connection status changes.
DeviceClient.preset_protocol_flags(...)	Preset server protocol flags.
DeviceClient.request(msg[, use_mid])	Send a request message, with automatic message ID assignment.
DeviceClient.running()	Whether the client is running.
DeviceClient.send_message(msg)	Send any kind of message.
DeviceClient.send_request(msg)	Send a request message.
DeviceClient.set_ioloop([ioloop])	Set the tornado.ioloop.IOLoop instance to use.
DeviceClient.start([timeout])	Start the client in a new thread.
DeviceClient.stop([timeout])	Stop a running client.
DeviceClient.unhandled_inform(msg)	Fallback method for inform messages without a registered handler.
DeviceClient.unhandled_reply(msg)	Fallback method for reply messages without a registered handler.
DeviceClient.unhandled_request(msg)	Fallback method for requests without a registered handler.
DeviceClient.until_connected(**kwargs)	Return future that resolves when the client is connected.
DeviceClient.until_protocol(**kwargs)	Return future that resolves after receipt of katcp protocol info.
DeviceClient.until_running([timeout])	Return future that resolves when the client is running.
DeviceClient.until_stopped([timeout])	Return future that resolves when the client has stopped.
DeviceClient.wait_connected([timeout])	Wait until the client is connected.
DeviceClient.wait_disconnected([timeout])	Wait until the client is disconnected.

Continued on next page

Table 33 – continued from previous page

DeviceClient.wait_protocol([timeout])	Wait until katcp protocol information has been received from server.
DeviceClient.wait_running([timeout])	Wait until the client is running.

MAX_LOOP_LATENCY = 0.03
Do not spend more than this many seconds reading pipelined socket data

IOStream inline-reading can result in ioloop starvation (see https://groups.google.com/forum/#!topic/python-tornado/yJrDAwDR_kA).

MAX_MSG_SIZE = 2097152
Maximum message size that can be received in bytes.

If more than MAX_MSG_SIZE bytes are read from the socket without encountering a message terminator (i.e. newline), the connection is closed.

MAX_WRITE_BUFFER_SIZE = 4194304
Maximum outstanding bytes to be buffered by the server process.

If more than MAX_WRITE_BUFFER_SIZE bytes are outstanding, the connection is closed. Note that the OS also buffers socket writes, so more than MAX_WRITE_BUFFER_SIZE bytes may be untransmitted in total.

bind_address
(host, port) where the client is connecting

convert_seconds(*time_seconds*)
Convert a time in seconds to the device timestamp units.

KATCP v4 and earlier, specified all timestamps in milliseconds. Since KATCP v5, all timestamps are in seconds. If the device KATCP version has been detected, this method converts a value in seconds to the appropriate (seconds or milliseconds) quantity. For version smaller than V4, the time value will be truncated to the nearest millisecond.

disconnect()
Force client connection to close, reconnect if auto-connect set.

enable_thread_safety()
Enable thread-safety features.

Must be called before start().

handle_inform(*msg*)
Dispatch an inform message to the appropriate method.

Parameters **msg** : Message object

The inform message to dispatch.

handle_message(*msg*)
Handle a message from the server.

Parameters **msg** : Message object

The Message to dispatch to the handler methods.

handle_reply(*msg*)
Dispatch a reply message to the appropriate method.

Parameters **msg** : Message object

The reply message to dispatch.

handle_request(*msg*)
Dispatch a request message to the appropriate method.

Parameters **msg** : Message object

The request message to dispatch.

inform_build_state(*msg*)
Handle katcp v4 and below build-state inform.

inform_version(*msg*)
Handle katcp v4 and below version inform.

inform_version_connect(*msg*)
Process a #version-connect message.

is_connected()
Check if the socket is currently connected.

Returns **connected** : bool

Whether the client is connected.

join(*timeout=None*)
Rejoin the client thread.

Parameters **timeout** : float in seconds

Seconds to wait for thread to finish.

Notes

Does nothing if the ioloop is not managed. Use until_stopped() instead.

notify_connected(*connected*)
Event handler that is called whenever the connection status changes.

Override in derived class for desired behaviour.

Note: This function should never block. Doing so will cause the client to cease processing data from the server until notify_connected completes.

Parameters **connected** : bool

Whether the client has just connected (True) or just disconnected (False).

preset_protocol_flags(*protocol_flags*)
Preset server protocol flags.

Sets the assumed server protocol flags and disables automatic server version detection.

Parameters **protocol_flags** : katcp.core.ProtocolFlags instance

request(*msg*, *use_mid=None*)
Send a request message, with automatic message ID assignment.

Parameters **msg** : katcp.Message request message

use_mid : bool or None, default=None

Returns **mid** : string or None

The message id, or None if no msg id is used

If use_mid is None and the server supports msg ids, or if use_mid is :

True a message ID will automatically be assigned msg.mid is None. :

if msg.mid has a value, and the server supports msg ids, that value :

will be used. If the server does not support msg ids, KatcpVersionError :

will be raised. :

running()
Whether the client is running.

Returns running : bool

Whether the client is running.

send_message(*msg*)
Send any kind of message.

Parameters msg : Message object

The message to send.

send_request(*msg*)
Send a request message.

Parameters msg : Message object

The request Message to send.

set_ioloop(*ioloop=None*)
Set the tornado.ioloop.IOLoop instance to use.

This defaults to IOLoop.current(). If set_ioloop() is never called the IOLoop is managed: started in a new thread, and will be stopped if self.stop() is called.

Notes

Must be called before start() is called

start(*timeout=None*)
Start the client in a new thread.

Parameters timeout : float in seconds

Seconds to wait for client thread to start. Do not specify a timeout if start() is being called from the same ioloop that this client will be installed on, since it will block the ioloop without progressing.

stop(*timeout=None*)
Stop a running client.

If using a managed ioloop, this must be called from a different thread to the ioloop's. This method only returns once the client's main coroutine, *_install()*, has completed.

If using an unmanaged ioloop, this can be called from the same thread as the ioloop. The *until_stopped()* method can be used to wait on completion of the main coroutine, *_install()*.

Parameters timeout : float in seconds

Seconds to wait for both client thread to have *started*, and for stopping.

unhandled_inform(*msg*)
Fallback method for inform messages without a registered handler.

Parameters **msg** : Message object

The inform message that wasn't processed by any handlers.

unhandled_reply(*msg*)
Fallback method for reply messages without a registered handler.

Parameters **msg** : Message object

The reply message that wasn't processed by any handlers.

unhandled_request(*msg*)
Fallback method for requests without a registered handler.

Parameters **msg** : Message object

The request message that wasn't processed by any handlers.

until_connected(***kwargs*)
Return future that resolves when the client is connected.

until_protocol(***kwargs*)
Return future that resolves after receipt of katcp protocol info.

If the returned future resolves, the server's protocol information is available in the ProtocolFlags instance self.protocol_flags.

until_running(*timeout=None*)
Return future that resolves when the client is running.

Notes

Must be called from the same ioloop as the client.

until_stopped(*timeout=None*)
Return future that resolves when the client has stopped.

Parameters **timeout** : float in seconds

Seconds to wait for the client to stop.

Notes

If already running, *stop()* must be called before this.

Must be called from the same ioloop as the client. If using a different thread, or a managed ioloop, this method should not be used. Use *join()* instead.

Also note that stopped != not running. Stopped means the main coroutine has ended, or was never started. When stopping, the running flag is cleared some time before stopped is set.

wait_connected(*timeout=None*)
Wait until the client is connected.

Parameters **timeout** : float in seconds

Seconds to wait for the client to connect.

Returns **connected** : bool

Whether the client is connected.

Notes

Do not call this from the ioloop, use until_connected().

wait_disconnected(*timeout=None*)
Wait until the client is disconnected.

Parameters **timeout** : float in seconds

Seconds to wait for the client to disconnect.

Returns **disconnected** : bool

Whether the client is disconnected.

Notes

Do not call this from the ioloop, use until_disconnected().

wait_protocol(*timeout=None*)
Wait until katcp protocol information has been received from server.

Parameters **timeout** : float in seconds

Seconds to wait for the client to connect.

Returns **received** : bool

Whether protocol information was received

If this method returns True, the server's protocol information is :

available in the ProtocolFlags instance self.protocol_flags. :

Notes

Do not call this from the ioloop, use until_protocol().

wait_running(*timeout=None*)
Wait until the client is running.

Parameters **timeout** : float in seconds

Seconds to wait for the client to start running.

Returns **running** : bool

Whether the client is running

Notes

Do not call this from the ioloop, use until_running().

katcp.client.**make_threadsafe**(*meth*)
Decorator for a DeviceClient method that should always run in ioloop.

Used with DeviceClient.enable_thread_safety(). If not called the method will be unprotected and it is the user's responsibility to ensure that these methods are only called from the ioloop, otherwise the decorated methods are wrapped. Should only be used for functions that have no return value.

katcp.client.**make_threadsafe_blocking**(*meth*)
Decorator for a DeviceClient method that will block.

Used with DeviceClient.enable_thread_safety(). Used to provide blocking calls that can be made from other threads. If called in ioloop context, calls the original method directly to prevent deadlocks. Will route return value to caller. Add *timeout* keyword argument to limit blocking time. If *meth* returns a future, its result will be returned, otherwise its result will be passed back directly.

katcp.client.**request_check**(*client*, *exception*, **msg_parms*, ***kwargs*)
Make blocking request to client and raise exception if reply is not ok.

Parameters **client** : DeviceClient instance

exception: Exception class to raise :

***msg_parms** : Message parameters sent to the Message.request() call

****kwargs** : Keyword arguments

Forwards kwargs['timeout'] to client.blocking_request(). Forwards kwargs['mid'] to Message.request().

Returns **reply, informs** : as returned by client.blocking_request

Raises ***exception* passed as parameter is raised if reply.reply_ok() is False** :

Notes

A typical use-case for this function is to use functools.partial() to bind a particular client and exception. The resulting function can then be used instead of direct client.blocking_request() calls to automate error handling.

1.5 Concrete Intermediate-level KATCP Client API (inspecting_client)

class katcp.inspecting_client.**ExponentialRandomBackoff**(*delay_initial=1.0*, *delay_max=90.0*, *exp_fac=3.0*, *randomicity=0.95*)
Bases: future.types.newobject.newobject

Methods

ExponentialRandomBackoff.failed()	Call whenever an action has failed, grows delay exponentially
ExponentialRandomBackoff.next()	
ExponentialRandomBackoff.success()	Call whenever an action has succeeded, resets delay to minimum

exp_fac = None
Increase timeout by this factor for each consecutive failure

failed()
Call whenever an action has failed, grows delay exponentially

After calling failed(), the *delay* property contains the next delay

success()

Call whenever an action has succeeded, resets delay to minimum

class katcp.inspecting_client.**InspectingClientAsync**(*host*, *port*, *ioloop=None*, *initial_inspection=None*, *auto_reconnect=True*, *logger=<logging.Logger object>*)

Bases: future.types.newobject.newobject

Higher-level client that inspects a KATCP interface.

Note: This class is not thread-safe at present, it should only be called from the ioloop.

Note: always call stop() after start() and you are done with the container to make sure the container cleans up correctly.

Methods

InspectingClientAsync.close()	
InspectingClientAsync.connect(**kwargs)	Connect to KATCP interface, starting what is needed
InspectingClientAsync.future_check_request(...)	Check if the request exists.
InspectingClientAsync.future_check_sensor(...)	Check if the sensor exists.
InspectingClientAsync.future_get_request(...)	Get the request object.
InspectingClientAsync.future_get_sensor(**kwargs)	Get the sensor object.
InspectingClientAsync.handle_sensor_value()	Handle #sensor-value informs just like #sensor-status informs
InspectingClientAsync.inform_hook_client_factory(...)	Return an instance of _InformHookDeviceClient or similar
InspectingClientAsync.inspect(**kwargs)	Inspect device requests and sensors, update model.
InspectingClientAsync.inspect_requests(**kwargs)	Inspect all or one requests on the device.
InspectingClientAsync.inspect_sensors(**kwargs)	Inspect all or one sensor on the device.
InspectingClientAsync.is_connected()	Connection status.
InspectingClientAsync.join([timeout])	
InspectingClientAsync.next()	
InspectingClientAsync.preset_protocol_flags(...)	Preset server protocol flags.
InspectingClientAsync.request_factory	Factory that produces KATCP Request objects.
InspectingClientAsync.sensor_factory	alias of katcp.core.Sensor
InspectingClientAsync.set_ioloop(ioloop)	
InspectingClientAsync.set_state_callback(cb)	Set user callback for state changes

Continued on next page

Table 35 – continued from previous page

InspectingClientAsync.simple_request(...)	Create and send a request to the server.
InspectingClientAsync.start([timeout])	Note: always call stop() and wait until_stopped() when you are done with the container to make sure the container cleans up correctly.
InspectingClientAsync.stop([timeout])	
InspectingClientAsync.until_connected([timeout])	
InspectingClientAsync.until_data_synced([...])	
InspectingClientAsync.until_not_synced([timeout])	
InspectingClientAsync.until_state(desired_state)	Wait until state is desired_state, InspectingClientStateType instance
InspectingClientAsync.until_stopped([timeout])	Return future that resolves when the client has stopped
InspectingClientAsync.until_synced([timeout])	
InspectingClientAsync.update_sensor(**kwargs)	

connect(***kwargs*)
Connect to KATCP interface, starting what is needed

Parameters **timeout** : float, None

Time to wait until connected. No waiting if None.

Raises **:class:'tornado.gen.TimeoutError' if the connect timeout expires** :

connected
Connection status.

future_check_request(***kwargs*)
Check if the request exists.

Used internally by future_get_request. This method is aware of synchronisation in progress and if inspection of the server is allowed.

Parameters **name** : str

Name of the request to verify.

update : bool or None, optional

If a katcp request to the server should be made to check if the sensor is on the server. True = Allow, False do not Allow, None use the class default.

Notes

Ensure that self.state.data_synced == True if yielding to future_check_request from a state-change callback, or a deadlock will occur.

future_check_sensor(***kwargs*)
Check if the sensor exists.

Used internally by future_get_sensor. This method is aware of synchronisation in progress and if inspection of the server is allowed.

Parameters **name** : str

Name of the sensor to verify.

update : bool or None, optional

If a katcp request to the server should be made to check if the sensor is on the server now.

Notes

Ensure that self.state.data_synced == True if yielding to future_check_sensor from a state-change callback, or a deadlock will occur.

future_get_request(***kwargs*)
Get the request object.

Check if we have information for this request, if not connect to server and update (if allowed).

Parameters **name** : string

Name of the request.

update : bool or None, optional

True allow inspect client to inspect katcp server if the request is not known.

Returns **Request created by :meth:‘request_factory‘ or None if request not found.** :

Notes

Ensure that self.state.data_synced == True if yielding to future_get_request from a state-change callback, or a deadlock will occur.

future_get_sensor(***kwargs*)
Get the sensor object.

Check if we have information for this sensor, if not connect to server and update (if allowed) to get information.

Parameters **name** : string

Name of the sensor.

update : bool or None, optional

True allow inspect client to inspect katcp server if the sensor is not known.

Returns **Sensor created by :meth:‘sensor_factory‘ or None if sensor not found.** :

Notes

Ensure that self.state.data_synced == True if yielding to future_get_sensor from a state-change callback, or a deadlock will occur.

handle_sensor_value()
Handle #sensor-value informs just like #sensor-status informs

inform_hook_client_factory(*host*, *port*, **args*, ***kwargs*)
Return an instance of `_InformHookDeviceClient` or similar

Provided to ease testing. Dynamically overriding this method after instantiation but before start() is called allows for deep brain surgery. See `katcp.fake_clients.TBD`

inspect(***kwargs*)
Inspect device requests and sensors, update model.

Returns **Tornado future that resolves with:** :

model_changes : Nested AttrDict or None

Contains sets of added/removed request/sensor names

Example structure:

```
{
    'requests':{
        'added': set(['req1', 'req2']),
        'removed': set(['req10', 'req20'])
    }
    'sensors': {
        'added': set(['sens1', 'sens2']),
        'removed': set(['sens10', 'sens20'])
    }
}
```

If there are no changes keys may be omitted. If an item is in both the 'added' and 'removed' sets that means that it changed.

If neither request not sensor changes are present, None is returned instead of a nested structure.

inspect_requests(***kwargs*)
Inspect all or one requests on the device. Update requests index.

Parameters **name** : str or None, optional

Name of the request or None to get all requests.

timeout : float or None, optional

Timeout for request inspection, None for no timeout

Returns **Tornado future that resolves with:** :

changes : `AttrDict`

AttrDict with keys `added` and `removed` (of type `set`), listing the requests that have been added or removed respectively. Modified requests are listed in both. If there are no changes, returns `None` instead.

Example structure:

```
{
    'added': set(['req1', 'req2']),
    'removed': set(['req10', 'req20'])
}
```

inspect_sensors(***kwargs*)
Inspect all or one sensor on the device. Update sensors index.

Parameters **name** : str or None, optional

Name of the sensor or None to get all sensors.

timeout : float or None, optional

Timeout for sensors inspection, None for no timeout

Returns **Tornado future that resolves with:** :

changes : `AttrDict`

AttrDict with keys `added` and `removed` (of type `set`), listing the sensors that have been added or removed respectively. Modified sensors are listed in both. If there are no changes, returns `None` instead.

Example structure:

```
{
    'added': set(['sens1', 'sens2']),
    'removed': set(['sens10', 'sens20'])
}
```

is_connected()
Connection status.

preset_protocol_flags(*protocol_flags*)
Preset server protocol flags.

Sets the assumed server protocol flags and disables automatic server version detection.

Parameters **protocol_flags** : katcp.core.ProtocolFlags instance

request_factory
Factory that produces KATCP Request objects.

signature: request_factory(name, description, timeout_hint), all parameters passed as kwargs

Should be set before calling connect()/start().

Methods

`Request.count`
`Request.index`

alias of `Request`

requests
A list of possible requests.

resync_delay = None
Set to an ExponentialRandomBackoff instance in _state_loop

sensor_factory
alias of `katcp.core.Sensor`

sensors
A list of known sensors.

set_state_callback(*cb*)
Set user callback for state changes

Called as `cb(state, model_changes)`

where *state* is an *InspectingClientStateType* instance, and *model_changes* is an AttrDict. The latter may contain keys requests and sensors to describe changes to requests or sensors respectively. These in turn have attributes added and removed which are sets of request/sensor names. Requests/sensors that have been modified will appear in both sets.

Warning: It is possible for *model_changes* to be None, or for either requests or sensors to be absent from *model_changes*.

simple_request(*request*, **args*, ***kwargs*)
Create and send a request to the server.

This method implements a very small subset of the options possible to send an request. It is provided as a shortcut to sending a simple request.

Parameters **request** : str

The request to call.

***args** : list of objects

Arguments to pass on to the request.

Keyword Arguments **timeout** : float or None, optional

Timeout after this amount of seconds (keyword argument).

mid : None or int, optional

Message identifier to use for the request message. If None, use either auto-incrementing value or no mid depending on the KATCP protocol version (mid's were only introduced with KATCP v5) and the value of the *use_mid* argument. Defaults to None

use_mid : bool

Use a mid for the request if True. Defaults to True if the server supports them.

Returns **future object.** :

Examples

```
reply, informs = yield ic.simple_request('help', 'sensor-list')
```

start(*timeout=None*)
Note: always call stop() and wait until_stopped() when you are done with the container to make sure the container cleans up correctly.

state
Current client state.

synced
Boolean indicating if the device has been synchronised.

until_state(*desired_state*, *timeout=None*)
Wait until state is desired_state, InspectingClientStateType instance

Returns a future

until_stopped(*timeout=None*)
Return future that resolves when the client has stopped

See the *DeviceClient.until_stopped* docstring for parameter definitions and more info.

class katcp.inspecting_client.**InspectingClientStateType**
Bases: *katcp.inspecting_client.InspectingClientStateType*

States tuple for the inspecting client. Fields, all bool:

connected: bool TCP connection has been established with the server.

synced: bool The inspecting client and the user that interfaces through the state change callback are all synchronised with the current device state. Also implies *connected* = *True* and *data_synced* = *True*.

model_changed: bool The device has changed in some way, resulting in the device model being out of date.

data_synced: bool The inspecting client's internal representation of the device is up to date, although state change user is not yet up to date.

Methods

InspectingClientStateType.count(value)	
InspectingClientStateType.index(value, ...)	Raises ValueError if the value is not present.

katcp.inspecting_client.**RequestType**
alias of katcp.inspecting_client.Request

exception katcp.inspecting_client.**SyncError**
Bases: exceptions.Exception

Raised if an error occurs during syncing with a device

1.6 Abstract High-level KATCP Client API (resource)

A high-level abstract interface to KATCP clients, sensors and requests.

class katcp.resource.**KATCPDummyRequest**(*request_description*, *is_active=<function <lambda>>*)
Bases: *katcp.resource.KATCPRequest*

Dummy counterpart to KATCPRequest that always returns a successful reply

Methods

KATCPDummyRequest.is_active()	True if resource for this request is active
KATCPDummyRequest.issue_request(*args, **kwargs)	Signature as for __call__
KATCPDummyRequest.next()	

issue_request(**args*, ***kwargs*)
Signature as for __call__

Do the request immediately without checking active state.

class katcp.resource.**KATCPReply**
Bases: katcp.resource._KATCPReplyTuple

Container for return messages of KATCP request (reply and informs).

This is based on a named tuple with 'reply' and 'informs' fields so that the *KATCPReply* object can still be unpacked into a normal tuple.

Parameters **reply** : *katcp.Message* object

Reply message returned by katcp request

informs : list of *katcp.Message* objects

List of inform messages returned by KATCP request

Attributes

messages: list of *katcp.Message* objects	List of all messages returned by KATCP request, reply first
reply: *katcp.Message* object	Reply message returned by KATCP request
informs: list of *katcp.Message* objects	List of inform messages returned by KATCP request
The instance evaluates to nonzero (i.e. truthy) if the request succeeded.	

Methods

KATCPReply.count(value)	
KATCPReply.index(value, [start, [stop]])	Raises ValueError if the value is not present.

messages
List of all messages returned by KATCP request, reply first.

succeeded
True if request succeeded (i.e. first reply argument is 'ok').

class katcp.resource.**KATCPRequest**(*request_description*, *is_active=<function <lambda>>*)
Bases: future.types.newobject.newobject

Abstract Base class to serve as the definition of the KATCPRequest API.

Wrapper around a specific KATCP request to a given KATCP device. Each available KATCP request for a particular device has an associated *KATCPRequest* object in the object hierarchy. This wrapper is mainly for interactive convenience. It provides the KATCP request help string as a docstring and pretty-prints the result of the request.

Methods

KATCPRequest.is_active()	True if resource for this request is active
KATCPRequest.issue_request(*args, **kwargs)	Signature as for __call__
KATCPRequest.next()	

description
Description of KATCP request as obtained from the ?help request.

is_active()
True if resource for this request is active

issue_request(**args*, ***kwargs*)
Signature as for __call__

Do the request immediately without checking active state.

name
Name of the KATCP request.

timeout_hint
Request timeout suggested by device or None if not provided

class katcp.resource.**KATCPResource**
Bases: future.types.newobject.newobject

Base class to serve as the definition of the KATCPResource API.

A class *C* implementing the KATCPResource API should register itself using KATCPResource.register(C) or subclass KATCPResource directly. A complication involved with subclassing is that all the abstract properties must be implemented as properties; normal instance attributes cannot be used.

Attributes

Apart from the abstract properties described below	
TODO Describe how hierarchies are implemented. Also all other descriptions	
here so that the sphinx doc can be auto-generated from here.	

Methods

KATCPResource.is_active()	
KATCPResource.list_sensors([filter, ...])	List sensors available on this resource matching certain criteria.
KATCPResource.next()	
KATCPResource.set_active(active)	
KATCPResource.set_sampling_strategies(**kwargs)	Set a sampling strategy for all sensors that match the specified filter.
KATCPResource.set_sampling_strategy(**kwargs)	Set a sampling strategy for a specific sensor.
KATCPResource.wait(**kwargs)	Wait for a sensor in this resource to satisfy a condition.

address
Address of the underlying client/device.

Type: tuple(host, port) or None, with host a string and port an integer.

If this KATCPResource is not associated with a specific KATCP device (e.g. it is only a top-level container for a hierarchy of KATCP resources), the address should be None.

children
AttrDict of subordinate KATCPResource objects keyed by their names.

description
Description of this KATCP resource.

is_connected
Indicate whether the underlying client/device is connected or not.

list_sensors (*filter=''*, *strategy=False*, *status=''*, *use_python_identifiers=True*, *tuple=False*, *refresh=False*)
List sensors available on this resource matching certain criteria.

Parameters **filter** : string, optional

Filter each returned sensor's name against this regexp if specified. To ease the dichotomy between Python identifier names and actual sensor names, the default is to search on Python identifier names rather than KATCP sensor names, unless *use_python_identifiers* below is set to False. Note that the sensors of subordinate KATCPResource instances may have inconsistent names and Python identifiers, better to always search on Python identifiers in this case.

strategy : {False, True}, optional

Only list sensors with a set strategy if True

status : string, optional

Filter each returned sensor's status against this regexp if given

use_python_identifiers : {True, False}, optional

Match on python identifiers even the the KATCP name is available.

tuple : {True, False}, optional, Default: False

Return backwards compatible tuple instead of SensorResultTuples

refresh : {True, False}, optional, Default: False

If set the sensor values will be refreshed with get_value before returning the results.

Returns **sensors** : list of SensorResultTuples, or list of tuples

List of matching sensors presented as named tuples. The *object* field is the `KATCPSensor` object associated with the sensor. Note that the name of the object may not match *name* if it originates from a subordinate device.

name
Name of this KATCP resource.

parent
Parent KATCPResource object of this subordinate resource, or None.

req
Attribute root/container for all KATCP request wrappers.

Each KATCP request that is exposed on a KATCP device should have a corresponding `KATCPRequest` object so that calling

resource.req.request_name(arg1, arg2, ...)

sends a '?request-name arg1 arg2 ...' message to the KATCP device and waits for the associated inform-reply and reply messages.

For a `KATCPResource` object that exposes a hierarchical device it can choose to include lower-level request handlers here such that *resource.req.dev_request()* maps to *resource.dev.req.request()*.

sensor
Attribute root/container for all KATCP sensor wrappers.

Each KATCP sensor that is exposed on a KATCP device should have a corresponding *KATCPSensor* object so that

resource.sensor.sensor_name

corresponds to a sensor named e.g. 'sensor-name', where the object or attribute name is an escaped/Pythonised version of the original sensor name (see *escape_name()* for the escape mechanism). Hopefully the device is not crazy enough to have multiple sensors that map to the same Python identifier.

A *KATCPResource* object that exposes a hierarchical device can choose to include lower-level sensors here such that *resource.sensor.dev_sensorname* maps to *resource.dev.sensor.sensorname*.

set_sampling_strategies(***kwargs*)

Set a sampling strategy for all sensors that match the specified filter.

Parameters **filter** : string

The regular expression filter to use to select the sensors to which to apply the specified strategy. Use "" to match all sensors. Is matched using *list_sensors()*.

strategy_and_params : seq of str or str

As tuple contains (<strat_name>, [<strat_parm1>, ...]) where the strategy names and parameters are as defined by the KATCP spec. As str contains the same elements in space-separated form.

****list_sensor_args** : keyword arguments

Passed to the *list_sensors()* call as kwargs

Returns **sensors_strategies** : tornado Future

resolves with a dict with the Python identifier names of the sensors as keys and the value a tuple:

(success, info) with

success [bool] True if setting succeeded for this sensor, else False

info [tuple] normalised sensor strategy and parameters as tuple if success == True else, sys.exc_info() tuple for the error that occurred.

set_sampling_strategy(***kwargs*)

Set a sampling strategy for a specific sensor.

Parameters **sensor_name** : string

The specific sensor.

strategy_and_params : seq of str or str

As tuple contains (<strat_name>, [<strat_parm1>, ...]) where the strategy names and parameters are as defined by the KATCP spec. As str contains the same elements in space-separated form.

Returns **sensors_strategies** : tornado Future

resolves with a dict with the Python identifier names of the sensors as keys and the value a tuple:

(success, info) with

success [bool] True if setting succeeded for this sensor, else False

info [tuple] normalised sensor strategy and parameters as tuple if success == True else, sys.exc_info() tuple for the error that occurred.

wait (***kwargs*)
Wait for a sensor in this resource to satisfy a condition.

Parameters **sensor_name** : string

The name of the sensor to check

condition_or_value : obj or callable, or seq of objs or callables

If obj, sensor.value is compared with obj. If callable, condition_or_value(reading) is called, and must return True if its condition is satisfied. Since the reading is passed in, the value, status, timestamp or received_timestamp attributes can all be used in the check.

timeout : float or None

The timeout in seconds (None means wait forever)

Returns **This command returns a tornado Future that resolves with True when the** :

sensor value satisfies the condition, or False if the condition is :

still not satisfied after a given timeout period. :

Raises **:class:'KATCPSensorError'** :

If the sensor does not have a strategy set, or if the named sensor is not present

exception katcp.resource.**KATCPResourceError**
Bases: exceptions.Exception

Error raised for resource-related errors

exception katcp.resource.**KATCPResourceInactive**
Bases: *katcp.resource.KATCPResourceError*

Raised when a request is made to an inactive resource

class katcp.resource.**KATCPSensor**(*sensor_description*, *sensor_manager*)
Bases: future.types.newobject.newobject

Wrapper around a specific KATCP sensor on a given KATCP device.

Each available KATCP sensor for a particular device has an associated *KATCPSensor* object in the object hierarchy. This wrapper is mainly for interactive convenience. It provides the KATCP request help string as a docstring and registers listeners. Subclasses need to call the base class version of __init__().

Methods

KATCPSensor.call_listeners(reading)	
KATCPSensor.clear_listeners()	Clear any registered listeners to updates from this sensor.
KATCPSensor.drop_sampling_strategy()	Drop memorised sampling strategy for sensor, if any
KATCPSensor.get_reading(**kwargs)	Get a fresh sensor reading from the KATCP resource
KATCPSensor.get_status(**kwargs)	Get a fresh sensor status from the KATCP resource
KATCPSensor.get_value(**kwargs)	Get a fresh sensor value from the KATCP resource
KATCPSensor.is_listener(listener)	
KATCPSensor.next()	
KATCPSensor.parse_value(s_value)	Parse a value from a string.

Continued on next page

Table 42 – continued from previous page

KATCPSensor.register_listener(listener[, ...])	Add a callback function that is called when sensor value is updated.
KATCPSensor.set(timestamp, status, value)	Set sensor with a given received value, matches *katcp.Sensor.set()*
KATCPSensor.set_formatted(raw_timestamp, ...)	Set sensor using KATCP string formatted inputs
KATCPSensor.set_sampling_strategy(strategy)	Set current sampling strategy for sensor
KATCPSensor.set_strategy(strategy[, params])	Set current sampling strategy for sensor.
KATCPSensor.set_value(value[, status, timestamp])	Set sensor value with optinal specification of status and timestamp
KATCPSensor.unregister_listener(listener)	Remove a listener callback added with register_listener().
KATCPSensor.wait(condition_or_value[, timeout])	Wait for the sensor to satisfy a condition.

clear_listeners()
Clear any registered listeners to updates from this sensor.

drop_sampling_strategy()
Drop memorised sampling strategy for sensor, if any

Calling this method ensures that the sensor manager does not attempt to reapply a sampling strategy. It will not raise an error if no strategy has been set. Use *set_sampling_strategy()* to memorise a strategy again.

get_reading(***kwargs*)
Get a fresh sensor reading from the KATCP resource

Returns **reply** : tornado Future resolving with *KATCPSensorReading* object

Notes

As a side-effect this will update the reading stored in this object, and result in registered listeners being called.

get_status(***kwargs*)
Get a fresh sensor status from the KATCP resource

Returns **reply** : tornado Future resolving with *KATCPSensorReading* object

Notes

As a side-effect this will update the reading stored in this object, and result in registered listeners being called.

get_value(***kwargs*)
Get a fresh sensor value from the KATCP resource

Returns **reply** : tornado Future resolving with *KATCPSensorReading* object

Notes

As a side-effect this will update the reading stored in this object, and result in registered listeners being called.

name
Name of this KATCPSensor

normalised_name
Normalised name of this KATCPSensor that can be used as a python identifier

parent_name
Name of the parent of this KATCPSensor

parse_value(*s_value*)
Parse a value from a string.

Parameters **s_value** : str

A string value to attempt to convert to a value for the sensor.

Returns **value** : object

A value of a type appropriate to the sensor.

reading
Most recently received sensor reading as KATCPSensorReading instance

register_listener(*listener*, *reading=False*)
Add a callback function that is called when sensor value is updated.

The callback footprint is received_timestamp, timestamp, status, value.

Parameters **listener** : function

Callback signature: if reading listener(katcp_sensor, reading) where *katcp_sensor* is this KATCPSensor instance *reading* is an instance of *KATCPSensorReading*.

Callback signature: default, if not reading listener(received_timestamp, timestamp, status, value)

sampling_strategy
Current sampling strategy

set(*timestamp*, *status*, *value*)
Set sensor with a given received value, matches *katcp.Sensor.set()*

set_formatted(*raw_timestamp*, *raw_status*, *raw_value*, *major*)
Set sensor using KATCP string formatted inputs

Mirrors *katcp.Sensor.set_formatted()*.

This implementation is empty. Will, during instantiation, be overridden by the set_formatted() method of a katcp.Sensor object.

set_sampling_strategy(*strategy*)
Set current sampling strategy for sensor

Parameters **strategy** : seq of str or str

As tuple contains (<strat_name>, [<strat_parm1>, ...]) where the strategy names and parameters are as defined by the KATCP spec. As str contains the same elements in space-separated form.

Returns **done** : tornado Future that resolves when done or raises KATCPSensorError

set_strategy(*strategy*, *params=None*)
Set current sampling strategy for sensor. Add this footprint for backwards compatibility.

Parameters **strategy** : seq of str or str

As tuple contains (<strat_name>, [<strat_parm1>, ...]) where the strategy names and parameters are as defined by the KATCP spec. As str contains the same elements in space-separated form.

params : seq of str or str

(<strat_name>, [<strat_parm1>, ...])

Returns **done** : tornado Future that resolves when done or raises KATCPSensorError

set_value(*value*, *status=1*, *timestamp=None*)
Set sensor value with optinal specification of status and timestamp

unregister_listener(*listener*)
Remove a listener callback added with register_listener().

Parameters **listener** : function

Reference to the callback function that should be removed

wait(*condition_or_value*, *timeout=None*)
Wait for the sensor to satisfy a condition.

Parameters **condition_or_value** : obj or callable, or seq of objs or callables

If obj, sensor.value is compared with obj. If callable, condition_or_value(reading) is called, and must return True if its condition is satisfied. Since the reading is passed in, the value, status, timestamp or received_timestamp attributes can all be used in the check. TODO: Sequences of conditions (use SensorTransitionWaiter thingum?)

timeout : float or None

The timeout in seconds (None means wait forever)

Returns **This command returns a tornado Future that resolves with True when the** :

sensor value satisfies the condition. It will never resolve with False; :

if a timeout is given a TimeoutError happens instead. :

Raises **:class:'KATCPSensorError'** :

If the sensor does not have a strategy set

:class:'tornado.gen.TimeoutError' :

If the sensor condition still fails after a stated timeout period

exception katcp.resource.**KATCPSensorError**
Bases: *katcp.resource.KATCPResourceError*

Raised if a problem occurred dealing with as KATCPSensor operation

class katcp.resource.**KATCPSensorReading**
Bases: *katcp.resource.KATCPSensorReading*

Sensor reading as a (received_timestamp, timestamp, istatus, value) tuple.

Attributes

KATCPSensorReading.received_timestamp	Alias for field number 0
KATCPSensorReading.timestamp	Alias for field number 1

Continued on next page

Table 43 – continued from previous page

KATCPSensorReading.istatus	Alias for field number 2
KATCPSensorReading.value	Alias for field number 3

Methods

KATCPSensorReading.count(value)	
KATCPSensorReading.index(value, [start, [stop]])	Raises ValueError if the value is not present.

status
Returns the string representation of sensor status, eg 'nominal'

class katcp.resource.**KATCPSensorsManager**
Bases: future.types.newobject.newobject

Sensor management class used by KATCPSensor. Abstracts communications details.

This class should arrange:

1. A mechanism for setting sensor strategies
2. A mechanism for polling a sensor value
3. Keeping track of- and reapplying sensor strategies after reconnect, etc.
4. Providing local time. This is doing to avoid direct calls to time.time, allowing accelerated time testing / simulation / dry-running

Methods

KATCPSensorsManager.drop_sampling_strategy(...)	Drop the sampling strategy for the named sensor from the cache
KATCPSensorsManager.get_sampling_strategy(...)	Get the current sampling strategy for the named sensor
KATCPSensorsManager.next()	
KATCPSensorsManager.poll_sensor(sensor_name)	Poll sensor and arrange for sensor object to be updated
KATCPSensorsManager.reapply_sampling_strategies()	Reapply all sensor strategies using cached values
KATCPSensorsManager.set_sampling_strategy(...)	Set the sampling strategy for the named sensor
KATCPSensorsManager.time()	Returns the current time (in seconds since UTC epoch)

drop_sampling_strategy(*sensor_name*)
Drop the sampling strategy for the named sensor from the cache

Calling *set_sampling_strategy()* requires the sensor manager to memorise the requested strategy so that it can automatically be reapplied. If the client is no longer interested in the sensor, or knows the sensor may be removed from the server, then it can use this method to ensure the manager forgets about the strategy. This method will not change the current strategy. No error is raised if there is no strategy to drop.

Parameters **sensor_name** : str

Name of the sensor (normal or escaped form)

get_sampling_strategy(*sensor_name*)
Get the current sampling strategy for the named sensor

Parameters **sensor_name** : str

Name of the sensor (normal or escaped form)

Returns **strategy** : tornado Future that resolves with tuple of str

contains (<strat_name>, [<strat_parm1>, ...]) where the strategy names and parameters are as defined by the KATCP spec

poll_sensor(*sensor_name*)
Poll sensor and arrange for sensor object to be updated

Returns **done_future** : tornado Future

Resolves when the poll is complete, or raises KATCPSensorError

reapply_sampling_strategies()
Reapply all sensor strategies using cached values

Would typically be called when a connection is re-established. Should not raise errors when resetting strategies for sensors that no longer exist on the KATCP resource.

set_sampling_strategy(*sensor_name*, *strategy_and_parms*)
Set the sampling strategy for the named sensor

Parameters **sensor_name** : str

Name of the sensor

strategy : seq of str or str

As tuple contains (<strat_name>, [<strat_parm1>, ...]) where the strategy names and parameters are as defined by the KATCP spec. As str contains the same elements in space-separated form.

Returns **done** : tornado Future that resolves when done or raises KATCPSensorError

Notes

It is recommended that implementations use *normalize_strategy_parameters()* to process the strategy_and_parms parameter, since it will deal with both string and list versions and makes sure that numbers are represented as strings in a consistent format.

This method should arrange for the strategy to be set on the underlying network device or whatever other implementation is used. This strategy should also be automatically re-set if the device is reconnected, etc. If a strategy is set for a non-existing sensor, it should still cache the strategy and ensure that is applied whenever said sensor comes into existence. This allows an applications to pre-set strategies for sensors before synced / connected to a device.

time()
Returns the current time (in seconds since UTC epoch)

class katcp.resource.**SensorResultTuple**
Bases: *katcp.resource.SensorResultTuple*

Per-sensor result of list_sensors() method

Attributes

SensorResultTuple.object	Alias for field number 0
SensorResultTuple.name	Alias for field number 1
SensorResultTuple.python_identifier	Alias for field number 2
SensorResultTuple.description	Alias for field number 3
SensorResultTuple.type	Alias for field number 4
SensorResultTuple.units	Alias for field number 5
SensorResultTuple.reading	Alias for field number 6

Methods

SensorResultTuple.count(value)	
SensorResultTuple.index(value, [start, [stop]])	Raises ValueError if the value is not present.

katcp.resource.**escape_name**(*name*)
Escape sensor and request names to be valid Python identifiers.

katcp.resource.**normalize_strategy_parameters**(*params*)
Normalize strategy parameters to be a list of strings.

Parameters **params** : (space-delimited) string or sequence of strings/numbers Parameters

expected by SampleStrategy object, in various forms, where the first parameter is the name of the strategy.

Returns **params** : tuple of strings

Strategy parameters as a list of strings

1.7 Concrete High-level KATCP Client API (resource_client)

class katcp.resource_client.**AttrMappingProxy**(*mapping*, *wrapper*)
Bases: *katcp.resource_client.MappingProxy*

Methods

AttrMappingProxy.get(k[,d])
AttrMappingProxy.items()
AttrMappingProxy.iteritems()
AttrMappingProxy.iterkeys()
AttrMappingProxy.itervalues()
AttrMappingProxy.keys()
AttrMappingProxy.values()

class katcp.resource_client.**ClientGroup**(*name*, *clients*)
Bases: future.types.newobject.newobject

Create a group of similar clients.

Parameters **name** : str

Name of the group of clients.

clients : list of `KATCPResource` objects

Clients to put into the group.

Methods

ClientGroup.client_updated(client)	Called to notify this group that a client has been updated.
ClientGroup.is_connected()	Indication of the connection state of all clients in the group
ClientGroup.next()	
ClientGroup.set_sampling_strategies(**kwargs)	Set sampling strategy for the sensors of all the group's clients.
ClientGroup.set_sampling_strategy(**kwargs)	Set sampling strategy for the sensors of all the group's clients.
ClientGroup.wait(**kwargs)	Wait for sensor present on all group clients to satisfy a condition.

client_updated(*client*)
Called to notify this group that a client has been updated.

is_connected()
Indication of the connection state of all clients in the group

set_sampling_strategies(***kwargs*)
Set sampling strategy for the sensors of all the group's clients.

Only sensors that match the specified filter are considered. See the *KATCPResource.set_sampling_strategies* docstring for parameter definitions and more info.

Returns **sensors_strategies** : tornado Future

Resolves with a dict with client names as keys and with the value as another dict. The value dict is similar to the return value described in the *KATCPResource.set_sampling_strategies* docstring.

set_sampling_strategy(***kwargs*)
Set sampling strategy for the sensors of all the group's clients.

Only sensors that match the specified filter are considered. See the *KATCPResource.set_sampling_strategies* docstring for parameter definitions and more info.

Returns **sensors_strategies** : tornado Future

Resolves with a dict with client names as keys and with the value as another dict. The value dict is similar to the return value described in the *KATCPResource.set_sampling_strategies* docstring.

wait(***kwargs*)
Wait for sensor present on all group clients to satisfy a condition.

Parameters **sensor_name** : string

The name of the sensor to check

condition_or_value : obj or callable, or seq of objs or callables

If obj, sensor.value is compared with obj. If callable, condition_or_value(reading) is called, and must return True if its condition is satisfied. Since the reading is passed in, the value, status, timestamp or received_timestamp attributes can all be used in the check.

timeout : float or None

The total timeout in seconds (None means wait forever)

quorum : None or int or float

The number of clients that are required to satisfy the condition, as either an explicit integer or a float between 0 and 1 indicating a fraction of the total number of clients, rounded up. If None, this means that all clients are required (the default). Be warned that a value of 1.0 (float) indicates all clients while a value of 1 (int) indicates a single client...

max_grace_period : float or None

After a quorum or initial timeout is reached, wait up to this long in an attempt to get the rest of the clients to satisfy condition as well (achieving effectively a full quorum if all clients behave)

Returns **This command returns a tornado Future that resolves with True when a** :

quorum of clients satisfy the sensor condition, or False if a quorum :

is not reached after a given timeout period (including a grace period). :

Raises **:class:'KATCPSensorError'** :

If any of the sensors do not have a strategy set, or if the named sensor is not present

class katcp.resource_client.**GroupRequest**(*group*, *name*, *description*)

Bases: future.types.newobject.newobject

Couroutine wrapper around a specific KATCP request for a group of clients.

Each available KATCP request supported by group has an associated *GroupRequest* object in the hierarchy. This wrapper is mainly for interactive convenience. It provides the KATCP request help string as a docstring accessible via IPython's question mark operator.

Parameters **Call parameters are all forwarded to the :class:'KATCPRequest' instance of each** :

client in the group. :

Returns **Returns a tornado future that resolves with a :class:'GroupResults' instance that** :

contains the replies of each client. If a particular client does not have the request, :

its result is None. :

Methods

GroupRequest.next()

class katcp.resource_client.**GroupResults**

Bases: dict

The result of a group request.

This has a dictionary interface, with the client names as keys and the corresponding replies from each client as

values. The replies are stored as `KATCPReply` objects, or are None for clients that did not support the request.

The result will evaluate to a truthy value if all the requests succeeded, i.e.

```
if result:
    handle_success()
else:
    handle_failure()
```

should work as expected.

Methods

`GroupResults.clear`()	
`GroupResults.copy`()	
`GroupResults.fromkeys`(S[,v])	v defaults to None.
`GroupResults.get`(k[,d])	
`GroupResults.has_key`(k)	
`GroupResults.items`()	
`GroupResults.iteritems`()	
`GroupResults.iterkeys`()	
`GroupResults.itervalues`()	
`GroupResults.keys`()	
`GroupResults.pop`(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
`GroupResults.popitem`()	2-tuple; but raise KeyError if D is empty.
`GroupResults.setdefault`(k[,d])	
`GroupResults.update`([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
`GroupResults.values`()	
`GroupResults.viewitems`()	
`GroupResults.viewkeys`()	
`GroupResults.viewvalues`()	

succeeded
: True if katcp request succeeded on all clients.

class katcp.resource_client.**KATCPClientResource**(*resource_spec*, *parent=None*, *logger=<logging.Logger object>*)

Bases: *katcp.resource.KATCPResource*

Class managing a client connection to a single KATCP resource

Inspects the KATCP interface of the resources, exposing sensors and requests as per the *katcp.resource.KATCPResource* API. Can also operate without exposing

Methods

KATCPClientResource.drop_sampling_strategy(...)	Drop the sampling strategy for the named sensor from the cache
KATCPClientResource.inspecting_client_factory(...)	Return an instance of *ReplyWrappedInspectingClientAsync* or similar
KATCPClientResource.is_active()	
KATCPClientResource.is_connected()	Indication of the connection state
KATCPClientResource.list_sensors([filter, ...])	List sensors available on this resource matching certain criteria.
KATCPClientResource.next()	
KATCPClientResource.set_active(active)	
KATCPClientResource.set_ioloop([ioloop])	Set the tornado ioloop to use
KATCPClientResource.set_sampling_strategies(...)	Set a strategy for all sensors matching the filter, including unseen sensors The strategy should persist across sensor disconnect/reconnect.
KATCPClientResource.set_sampling_strategy(...)	Set a strategy for a sensor even if it is not yet known.
KATCPClientResource.set_sensor_listener(**kwargs)	Set a sensor listener for a sensor even if it is not yet known The listener registration should persist across sensor disconnect/reconnect.
KATCPClientResource.start()	Start the client and connect
KATCPClientResource.stop()	
KATCPClientResource.until_not_synced([timeout])	Convenience method to wait (with Future) until client is not synced
KATCPClientResource.until_state(state[, timeout])	Future that resolves when a certain client state is attained
KATCPClientResource.until_stopped([timeout])	Return future that resolves when the inspecting client has stopped
KATCPClientResource.until_synced([timeout])	Convenience method to wait (with Future) until client is synced
KATCPClientResource.wait(**kwargs)	Wait for a sensor in this resource to satisfy a condition.
KATCPClientResource.wait_connected([timeout])	Future that resolves when the state is not 'disconnected'.

MAX_LOOP_LATENCY = 0.03

When doing potentially tight loops in coroutines yield tornado.gen.moment after this much time. This is a suggestion for methods to use.

drop_sampling_strategy(*sensor_name*)

Drop the sampling strategy for the named sensor from the cache

Calling *set_sampling_strategy()* requires the requested strategy to be memorised so that it can automatically be reapplied. This method causes the strategy to be forgotten. There is no change to the current strategy. No error is raised if there is no strategy to drop.

Parameters **sensor_name** : str

Name of the sensor

inspecting_client_factory(*host*, *port*, *ioloop_set_to*)

Return an instance of *ReplyWrappedInspectingClientAsync* or similar

Provided to ease testing. Dynamically overriding this method after instantiation but before start() is called allows for deep brain surgery. See `katcp.fake_clients.fake_inspecting_client_factory`

is_connected()
Indication of the connection state

Returns True if state is not "disconnected", i.e "syncing" or "synced"

list_sensors(*filter=''*, *strategy=False*, *status=''*, *use_python_identifiers=True*, *tuple=False*, *refresh=False*)
List sensors available on this resource matching certain criteria.

Parameters **filter** : string, optional

Filter each returned sensor's name against this regexp if specified. To ease the dichotomy between Python identifier names and actual sensor names, the default is to search on Python identifier names rather than KATCP sensor names, unless *use_python_identifiers* below is set to False. Note that the sensors of subordinate KATCPResource instances may have inconsistent names and Python identifiers, better to always search on Python identifiers in this case.

strategy : {False, True}, optional

Only list sensors with a set strategy if True

status : string, optional

Filter each returned sensor's status against this regexp if given

use_python_identifiers : {True, False}, optional

Match on python identifiers even the the KATCP name is available.

tuple : {True, False}, optional, Default: False

Return backwards compatible tuple instead of SensorResultTuples

refresh : {True, False}, optional, Default: False

If set the sensor values will be refreshed with get_value before returning the results.

Returns **sensors** : list of SensorResultTuples, or list of tuples

List of matching sensors presented as named tuples. The *object* field is the `KATCPSensor` object associated with the sensor. Note that the name of the object may not match *name* if it originates from a subordinate device.

set_ioloop(*ioloop=None*)
Set the tornado ioloop to use

Defaults to tornado.ioloop.IOLoop.current() if set_ioloop() is not called or if ioloop=None. Must be called before start()

set_sampling_strategies(***kwargs*)
Set a strategy for all sensors matching the filter, including unseen sensors The strategy should persist across sensor disconnect/reconnect.

filter [str] Filter for sensor names

strategy_and_params [seq of str or str] As tuple contains (<strat_name>, [<strat_parm1>, ...]) where the strategy names and parameters are as defined by the KATCP spec. As str contains the same elements in space-separated form.

Returns **done** : tornado Future

Resolves when done

set_sampling_strategy(***kwargs*)
Set a strategy for a sensor even if it is not yet known. The strategy should persist across sensor disconnect/reconnect.

sensor_name [str] Name of the sensor

strategy_and_params [seq of str or str] As tuple contains (<strat_name>, [<strat_parm1>, ...]) where the strategy names and parameters are as defined by the KATCP spec. As str contains the same elements in space-separated form.

Returns **done** : tornado Future

Resolves when done

set_sensor_listener(***kwargs*)
Set a sensor listener for a sensor even if it is not yet known The listener registration should persist across sensor disconnect/reconnect.

sensor_name [str] Name of the sensor

listener [callable] Listening callable that will be registered on the named sensor when it becomes available. Callable as for `KATCPSensor.register_listener()`

start()
Start the client and connect

until_not_synced(*timeout=None*)
Convenience method to wait (with Future) until client is not synced

until_state(*state*, *timeout=None*)
Future that resolves when a certain client state is attained

Parameters **state** : str

Desired state, one of ("disconnected", "syncing", "synced")

timeout: float :

Timeout for operation in seconds.

until_stopped(*timeout=None*)
Return future that resolves when the inspecting client has stopped

See the *DeviceClient.until_stopped* docstring for parameter definitions and more info.

until_synced(*timeout=None*)
Convenience method to wait (with Future) until client is synced

wait_connected(*timeout=None*)
Future that resolves when the state is not 'disconnected'.

class katcp.resource_client.**KATCPClientResourceContainer**(*resources_spec*, *logger=<logging.Logger object>*)

Bases: *katcp.resource.KATCPResource*

Class for containing multiple *KATCPClientResource* instances.

Provides aggregate *sensor* and *req* attributes containing the union of all the sensors in requests in the contained resources. Names are prefixed with <resname>_, where <resname> is the name of the resource to which the sensor / request belongs except for aggregate sensors that starts with `agg_`.

Methods

KATCPClientResourceContainer.add_child_resource_client(...)	Add a resource client to the container and start the resource connection
KATCPClientResourceContainer.add_group(...)	Add a new *ClientGroup* to container groups member.
KATCPClientResourceContainer.client_resource_factory(...)	Return an instance of *KATCPClientResource* or similar
KATCPClientResourceContainer.is_active()	
KATCPClientResourceContainer.is_connected()	Indication of the connection state of all children
KATCPClientResourceContainer.list_sensors([...])	List sensors available on this resource matching certain criteria.
KATCPClientResourceContainer.next()	
KATCPClientResourceContainer.set_active(active)	
KATCPClientResourceContainer.set_ioloop([ioloop])	Set the tornado ioloop to use
KATCPClientResourceContainer.set_sampling_strategies(...)	Set sampling strategies for filtered sensors - these sensors have to exists.
KATCPClientResourceContainer.set_sampling_strategy(...)	Set sampling strategies for the specific sensor - this sensor has to exist
KATCPClientResourceContainer.set_sensor_listener(...)	Set listener for the specific sensor - this sensor has to exists.
KATCPClientResourceContainer.start()	Start and connect all the subordinate clients
KATCPClientResourceContainer.stop()	Stop all child resources
KATCPClientResourceContainer.until_all_children_in_state(...)	Return a tornado Future; resolves when all clients are in specified state
KATCPClientResourceContainer.until_any_child_in_state(state)	Return a tornado Future; resolves when any client is in specified state
KATCPClientResourceContainer.until_not_synced(...)	Return a tornado Future; resolves when any subordinate client is not synced
KATCPClientResourceContainer.until_stopped([...])	Return dict of futures that resolve when each child resource has stopped
KATCPClientResourceContainer.until_synced(...)	Return a tornado Future; resolves when all subordinate clients are synced
KATCPClientResourceContainer.wait(... [, timeout])	Wait for a sensor in this resource to satisfy a condition.

add_child_resource_client(*res_name*, *res_spec*)
Add a resource client to the container and start the resource connection

add_group(*group_name*, *group_client_names*)
Add a new *ClientGroup* to container groups member.

Add the group named *group_name* with sequence of client names to the container groups member. From there it will be wrapped appropriately in the higher-level thread-safe container.

client_resource_factory(*res_spec*, *parent*, *logger*)
Return an instance of *KATCPClientResource* or similar

Provided to ease testing. Overriding this method allows deep brain surgery. See katcp.

```
fake_clients.fake_KATCP_client_resource_factory()
```

is_connected()
Indication of the connection state of all children

list_sensors(*filter=''*, *strategy=False*, *status=''*, *use_python_identifiers=True*, *tuple=False*, *refresh=False*)
List sensors available on this resource matching certain criteria.

Parameters **filter** : string, optional

Filter each returned sensor's name against this regexp if specified. To ease the dichotomy between Python identifier names and actual sensor names, the default is to search on Python identifier names rather than KATCP sensor names, unless *use_python_identifiers* below is set to False. Note that the sensors of subordinate KATCPResource instances may have inconsistent names and Python identifiers, better to always search on Python identifiers in this case.

strategy : {False, True}, optional

Only list sensors with a set strategy if True

status : string, optional

Filter each returned sensor's status against this regexp if given

use_python_identifiers : {True, False}, optional

Match on python identifiers even the the KATCP name is available.

tuple : {True, False}, optional, Default: False

Return backwards compatible tuple instead of SensorResultTuples

refresh : {True, False}, optional, Default: False

If set the sensor values will be refreshed with get_value before returning the results.

Returns **sensors** : list of SensorResultTuples, or list of tuples

List of matching sensors presented as named tuples. The *object* field is the `KATCPSensor` object associated with the sensor. Note that the name of the object may not match *name* if it originates from a subordinate device.

set_ioloop(*ioloop=None*)
Set the tornado ioloop to use

Defaults to tornado.ioloop.IOLoop.current() if set_ioloop() is not called or if ioloop=None. Must be called before start()

set_sampling_strategies(***kwargs*)
Set sampling strategies for filtered sensors - these sensors have to exists.

set_sampling_strategy(***kwargs*)
Set sampling strategies for the specific sensor - this sensor has to exist

set_sensor_listener(***kwargs*)
Set listener for the specific sensor - this sensor has to exists.

start()
Start and connect all the subordinate clients

stop()
Stop all child resources

until_all_children_in_state(***kwargs*)
Return a tornado Future; resolves when all clients are in specified state

until_any_child_in_state(*state*, *timeout=None*)
Return a tornado Future; resolves when any client is in specified state

until_not_synced(***kwargs*)
Return a tornado Future; resolves when any subordinate client is not synced

until_stopped(*timeout=None*)
Return dict of futures that resolve when each child resource has stopped

See the *DeviceClient.until_stopped* docstring for parameter definitions and more info.

until_synced(***kwargs*)
Return a tornado Future; resolves when all subordinate clients are synced

wait(*sensor_name*, *condition_or_value*, *timeout=5*)
Wait for a sensor in this resource to satisfy a condition.

Parameters **sensor_name** : string

The name of the sensor to check

condition_or_value : obj or callable, or seq of objs or callables

If obj, sensor.value is compared with obj. If callable, condition_or_value(reading) is called, and must return True if its condition is satisfied. Since the reading is passed in, the value, status, timestamp or received_timestamp attributes can all be used in the check.

timeout : float or None

The timeout in seconds (None means wait forever)

Returns **This command returns a tornado Future that resolves with True when the** :

sensor value satisfies the condition, or False if the condition is :

still not satisfied after a given timeout period. :

Raises **:class:'KATCPSensorError'** :

If the sensor does not have a strategy set, or if the named sensor is not present

class katcp.resource_client.**KATCPClientResourceRequest**(*request_description*, *client*, *is_active=<function <lambda>>*)

Bases: *katcp.resource.KATCPRequest*

Callable wrapper around a KATCP request

Methods

KATCPClientResourceRequest.is_active()	True if resource for this request is active
KATCPClientResourceRequest.issue_request(...)	Issue the wrapped request to the server.
KATCPClientResourceRequest.next()	

issue_request(**args*, ***kwargs*)
Issue the wrapped request to the server.

Parameters ***args** : list of objects

Arguments to pass on to the request.

Keyword Arguments **timeout** : float or None, optional

Timeout after this amount of seconds (keyword argument).

mid : None or int, optional

Message identifier to use for the request message. If None, use either auto-incrementing value or no mid depending on the KATCP protocol version (mid's were only introduced with KATCP v5) and the value of the *use_mid* argument. Defaults to None.

use_mid : bool

Use a mid for the request if True.

Returns **future object that resolves with an :class:'katcp.resource.KATCPReply'** :

instance :

class katcp.resource_client.**KATCPClientResourceSensorsManager**(*inspecting_client*, *resource_name*, *logger=<logging.Logger object>*)

Bases: `future.types.newobject.newobject`

Implementation of KATSensorsManager ABC for a directly-connected client

Assumes that all methods are called from the same ioloop context

Methods

KATCPClientResourceSensorsManager.drop_sampling_strategy(...)	Drop the sampling strategy for the named sensor from the cache
KATCPClientResourceSensorsManager.get_sampling_strategy(...)	Get the current sampling strategy for the named sensor
KATCPClientResourceSensorsManager.next()	
KATCPClientResourceSensorsManager.poll_sensor(...)	Poll sensor and arrange for sensor object to be updated
KATCPClientResourceSensorsManager.reapply_sampling_strategies(...)	Reapply all sensor strategies using cached values
KATCPClientResourceSensorsManager.sensor_factory(...)	
KATCPClientResourceSensorsManager.set_sampling_strategy(...)	Set the sampling strategy for the named sensor

drop_sampling_strategy(*sensor_name*)

Drop the sampling strategy for the named sensor from the cache

Calling *set_sampling_strategy()* requires the sensor manager to memorise the requested strategy so that it can automatically be reapplied. If the client is no longer interested in the sensor, or knows the sensor may be removed from the server, then it can use this method to ensure the manager forgets about the strategy. This method will not change the current strategy. No error is raised if there is no strategy to drop.

Parameters **sensor_name** : str

Name of the sensor (normal or escaped form)

get_sampling_strategy(*sensor_name*)
Get the current sampling strategy for the named sensor

Parameters **sensor_name** : str

Name of the sensor (normal or escaped form)

Returns **strategy** : tuple of str

contains (<strat_name>, [<strat_parm1>, ...]) where the strategy names and parameters are as defined by the KATCP spec

poll_sensor(***kwargs*)
Poll sensor and arrange for sensor object to be updated

Returns **done_future** : tornado Future

Resolves when the poll is complete, or raises KATCPSensorError

reapply_sampling_strategies(***kwargs*)
Reapply all sensor strategies using cached values

set_sampling_strategy(***kwargs*)
Set the sampling strategy for the named sensor

Parameters **sensor_name** : str

Name of the sensor

strategy_and_params : seq of str or str

As tuple contains (<strat_name>, [<strat_parm1>, ...]) where the strategy names and parameters are as defined by the KATCP spec. As str contains the same elements in space-separated form.

Returns **sensor_strategy** : tuple

(success, info) with

success [bool] True if setting succeeded for this sensor, else False

info [tuple] Normalised sensor strategy and parameters as tuple if success == True else, sys.exc_info() tuple for the error that occurred.

class katcp.resource_client.**MappingProxy**(*mapping*, *wrapper*)
Bases: _abcoll.Mapping

Methods

MappingProxy.get(k[,d])
MappingProxy.items()
MappingProxy.iteritems()
MappingProxy.iterkeys()
MappingProxy.itervalues()
MappingProxy.keys()
MappingProxy.values()

class katcp.resource_client.**ReplyWrappedInspectingClientAsync**(*host*, *port*, *ioloop=None*, *initial_inspection=None*, *auto_reconnect=True*, *logger=<logging.Logger object>*)

Bases: *katcp.inspecting_client.InspectingClientAsync*

Adds wrapped_request() method that wraps reply in a KATCPReply

Methods

ReplyWrappedInspectingClientAsync.close()	
ReplyWrappedInspectingClientAsync.connect(...)	Connect to KATCP interface, starting what is needed
ReplyWrappedInspectingClientAsync.future_check_request(...)	Check if the request exists.
ReplyWrappedInspectingClientAsync.future_check_sensor(...)	Check if the sensor exists.
ReplyWrappedInspectingClientAsync.future_get_request(...)	Get the request object.
ReplyWrappedInspectingClientAsync.future_get_sensor(...)	Get the sensor object.
ReplyWrappedInspectingClientAsync.handle_sensor_value()	Handle #sensor-value informs just like #sensor-status informs
ReplyWrappedInspectingClientAsync.inform_hook_client_factory(...)	Return an instance of _InformHookDeviceClient or similar
ReplyWrappedInspectingClientAsync.inspect(...)	Inspect device requests and sensors, update model.
ReplyWrappedInspectingClientAsync.inspect_requests(...)	Inspect all or one requests on the device.
ReplyWrappedInspectingClientAsync.inspect_sensors(...)	Inspect all or one sensor on the device.
ReplyWrappedInspectingClientAsync.is_connected()	Connection status.
ReplyWrappedInspectingClientAsync.join([timeout])	
ReplyWrappedInspectingClientAsync.next()	
ReplyWrappedInspectingClientAsync.preset_protocol_flags(...)	Preset server protocol flags.
ReplyWrappedInspectingClientAsync.reply_wrapper(x)	
ReplyWrappedInspectingClientAsync.request_factory	alias of katcp.inspecting_client.Request
ReplyWrappedInspectingClientAsync.sensor_factory	alias of katcp.core.Sensor
ReplyWrappedInspectingClientAsync.set_ioloop(ioloop)	

Continued on next page

Table 57 – continued from previous page

ReplyWrappedInspectingClientAsync.set_state_callback(cb)	Set user callback for state changes
ReplyWrappedInspectingClientAsync.simple_request(...)	Create and send a request to the server.
ReplyWrappedInspectingClientAsync.start([...])	Note: always call stop() and wait until_stopped() when you are done with the container to make sure the container cleans up correctly.
ReplyWrappedInspectingClientAsync.stop([timeout])	
ReplyWrappedInspectingClientAsync.until_connected([...])	
ReplyWrappedInspectingClientAsync.until_data_synced([...])	
ReplyWrappedInspectingClientAsync.until_not_synced([...])	
ReplyWrappedInspectingClientAsync.until_state(...)	Wait until state is desired_state, InspectingClientStateType instance
ReplyWrappedInspectingClientAsync.until_stopped([...])	Return future that resolves when the client has stopped
ReplyWrappedInspectingClientAsync.until_synced([...])	
ReplyWrappedInspectingClientAsync.update_sensor(...)	
ReplyWrappedInspectingClientAsync.wrapped_request(...)	Create and send a request to the server.

wrapped_request (*request*, **args*, ***kwargs*)

Create and send a request to the server.

This method implements a very small subset of the options possible to send an request. It is provided as a shortcut to sending a simple wrapped request.

Parameters **request** : str

The request to call.

***args** : list of objects

Arguments to pass on to the request.

Keyword Arguments **timeout** : float or None, optional

Timeout after this amount of seconds (keyword argument).

mid : None or int, optional

Message identifier to use for the request message. If None, use either auto-incrementing value or no mid depending on the KATCP protocol version (mid's were only introduced with KATCP v5) and the value of the *use_mid* argument. Defaults to None.

use_mid : bool

Use a mid for the request if True.

Returns **future object that resolves with the** :

:meth:'katcp.client.DeviceClient.future_request' response wrapped in :

self.reply_wrapper :

class katcp.resource_client.**ThreadSafeKATCPClientGroupWrapper**(*subject*, *ioloop_wrapper*)

Bases: katcp.ioloop_manager.ThreadSafeMethodAttrWrapper

Thread safe wrapper for *ClientGroup*

Methods

ThreadSafeKATCPClientGroupWrapper. next()

class katcp.resource_client.**ThreadSafeKATCPClientResourceRequestWrapper**(*subject*, *ioloop_wrapper*)

Bases: katcp.ioloop_manager.ThreadSafeMethodAttrWrapper

Methods

ThreadSafeKATCPClientResourceRequestWrapper. next()

class katcp.resource_client.**ThreadSafeKATCPClientResourceWrapper**(*subject*, *ioloop_wrapper*)

Bases: katcp.ioloop_manager.ThreadSafeMethodAttrWrapper

Should work with both KATCPClientResource or KATCPClientResourceContainer

Methods

ThreadSafeKATCPClientResourceWrapper. next()

class katcp.resource_client.**ThreadSafeKATCPSensorWrapper**(*subject*, *ioloop_wrapper*)

Bases: katcp.ioloop_manager.ThreadSafeMethodAttrWrapper

Methods

ThreadSafeKATCPSensorWrapper.next()

katcp.resource_client.**list_sensors**(**args*, ***kwargs*)

Helper for implementing *katcp.resource.KATCPResource.list_sensors()*

Parameters **sensor_items** : tuple of sensor-item tuples

As would be returned the items() method of a dict containing KATCPSensor objects keyed by Python-identifiers.

parent_class: KATCPClientResource or KATCPClientResourceContainer :

Is used for prefix calculation

Rest of parameters as for :meth:'katcp.resource.KATCPResource.list_sensors' :

katcp.resource_client.**monitor_resource_sync_state**(**args*, ***kwargs*)

Coroutine that monitors a KATCPResource's sync state.

Calls callback(True/False) whenever the resource becomes synced or unsynced. Will always do an initial callback(False) call. Exits without calling callback() if exit_event is set.

Warning: set the monitor's exit_event before stopping the resources being monitored, otherwise it could result in a memory leak. The *until_synced()* or *until_not_synced()* methods could keep a reference to the resource alive.

katcp.resource_client.**transform_future**(*transformation*, *future*)

Returns a new future that will resolve with a transformed value.

Takes the resolution value of *future* and applies "transformation(*future.result())" to it before setting the result of the new future with the transformed value. If future() resolves with an exception, it is passed through to the new future.

Assumes *future* is a tornado Future.

1.8 Sampling

Strategies for sampling sensor values.

class katcp.sampling.**SampleAuto**(*inform_callback*, *sensor*, **params*, ***kwargs*)

Bases: *katcp.sampling.SampleStrategy*

Strategy which sends updates whenever the sensor itself is updated.

Methods

SampleAuto.attach()	Attach strategy to its sensor and send initial update.
SampleAuto.cancel()	Detach strategy from its sensor and cancel ioloop callbacks.
SampleAuto.cancel_timeouts()	Override this method to cancel any outstanding ioloop timeouts.
SampleAuto.detach()	Detach strategy from its sensor.
SampleAuto.get_sampling()	The Strategy constant for this sampling strategy.
SampleAuto.get_sampling_formatted()	The current sampling strategy and parameters.
SampleAuto.get_strategy(strategyName, ...)	Factory method to create a strategy object.
SampleAuto.inform(reading)	Inform strategy creator of the sensor status.
SampleAuto.start()	Start operating the strategy.
SampleAuto.update(sensor, reading)	Callback used by the sensor's notify() method.

get_sampling()

The Strategy constant for this sampling strategy.

Sub-classes should implement this method and return the appropriate constant.

Returns **strategy** : Strategy constant

The strategy type constant for this strategy.

update(*sensor*, *reading*)

Callback used by the sensor's notify() method.

This update method is called whenever the sensor value is set so sensor will contain the right info. Note that the strategy does not really need to be passed a sensor because it already has a handle to it, but receives it due to the generic observer mechanism.

Sub-classes should override this method or `start()` to provide the necessary sampling strategy. Sub-classes should also ensure that *`update()`* is thread-safe; an easy way to do this is by using the @update_in_ioloop decorator.

Parameters **sensor** : Sensor object

The sensor which was just updated.

reading : (timestamp, status, value) tuple

Sensor reading as would be returned by sensor.read()

class katcp.sampling.**SampleDifferential**(*inform_callback*, *sensor*, **params*, ***kwargs*)

Bases: *katcp.sampling.SampleStrategy*

Differential sampling strategy for integer and float sensors.

Sends updates only when the value has changed by more than some specified threshold, or the status changes.

Methods

SampleDifferential.attach()	Attach strategy to its sensor and send initial update.
SampleDifferential.cancel()	Detach strategy from its sensor and cancel ioloop callbacks.
SampleDifferential.cancel_timeouts()	Override this method to cancel any outstanding ioloop timeouts.
SampleDifferential.detach()	Detach strategy from its sensor.
SampleDifferential.get_sampling()	The Strategy constant for this sampling strategy.
SampleDifferential.get_sampling_formatted()	The current sampling strategy and parameters.
SampleDifferential.get_strategy(...)	Factory method to create a strategy object.
SampleDifferential.inform(reading)	Inform strategy creator of the sensor status.
SampleDifferential.start()	Start operating the strategy.
SampleDifferential.update(sensor, reading)	Callback used by the sensor's notify() method.

get_sampling()

The Strategy constant for this sampling strategy.

Sub-classes should implement this method and return the appropriate constant.

Returns **strategy** : Strategy constant

The strategy type constant for this strategy.

update(*sensor*, *reading*)

Callback used by the sensor's notify() method.

This update method is called whenever the sensor value is set so sensor will contain the right info. Note that the strategy does not really need to be passed a sensor because it already has a handle to it, but receives it due to the generic observer mechanism.

Sub-classes should override this method or `start()` to provide the necessary sampling strategy. Sub-

classes should also ensure that *update()* is thread-safe; an easy way to do this is by using the @update_in_ioloop decorator.

Parameters **sensor** : Sensor object

The sensor which was just updated.

reading : (timestamp, status, value) tuple

Sensor reading as would be returned by sensor.read()

class katcp.sampling.**SampleDifferentialRate**(*inform_callback*, *sensor*, **params*, ***kwargs*)

Bases: *katcp.sampling.SampleEventRate*

Differential rate sampling strategy.

Report the value whenever it changes by more than *difference* from the last reported value or if more than *longest_period* seconds have passed since the last reported update. However, do not report the value until *shortest_period* seconds have passed since the last reported update. The behaviour if *shortest_period* is greater than *longest_period* is undefined. May only be implemented for float and integer sensors.

Methods

SampleDifferentialRate.attach()	Attach strategy to its sensor and send initial update.
SampleDifferentialRate.cancel()	Detach strategy from its sensor and cancel ioloop callbacks.
SampleDifferentialRate.cancel_timeouts()	Override this method to cancel any outstanding ioloop timeouts.
SampleDifferentialRate.detach()	Detach strategy from its sensor.
SampleDifferentialRate.get_sampling()	The Strategy constant for this sampling strategy.
SampleDifferentialRate.get_sampling_formatted()	The current sampling strategy and parameters.
SampleDifferentialRate.get_strategy(...)	Factory method to create a strategy object.
SampleDifferentialRate.inform(reading)	Inform strategy creator of the sensor status.
SampleDifferentialRate.start()	Start operating the strategy.
SampleDifferentialRate.update(sensor, reading)	Callback used by the sensor's notify() method.

get_sampling()

The Strategy constant for this sampling strategy.

Sub-classes should implement this method and return the appropriate constant.

Returns **strategy** : Strategy constant

The strategy type constant for this strategy.

class katcp.sampling.**SampleEvent**(*inform_callback*, *sensor*, **params*, ***kwargs*)

Bases: *katcp.sampling.SampleEventRate*

Strategy which sends updates when the sensor value or status changes.

Since SampleEvent is just a special case of SampleEventRate, we use SampleEventRate with the appropriate default values to implement SampleEvent.

Methods

SampleEvent.attach()	Attach strategy to its sensor and send initial update.
SampleEvent.cancel()	Detach strategy from its sensor and cancel ioloop callbacks.
SampleEvent.cancel_timeouts()	Override this method to cancel any outstanding ioloop timeouts.
SampleEvent.detach()	Detach strategy from its sensor.
SampleEvent.get_sampling()	The Strategy constant for this sampling strategy.
SampleEvent.get_sampling_formatted()	The current sampling strategy and parameters.
SampleEvent.get_strategy(strategyName, ...)	Factory method to create a strategy object.
SampleEvent.inform(reading)	Inform strategy creator of the sensor status.
SampleEvent.start()	Start operating the strategy.
SampleEvent.update(sensor, reading)	Callback used by the sensor's notify() method.

get_sampling()
The Strategy constant for this sampling strategy.

Sub-classes should implement this method and return the appropriate constant.

Returns **strategy** : Strategy constant

The strategy type constant for this strategy.

class katcp.sampling.**SampleEventRate**(*inform_callback*, *sensor*, **params*, ***kwargs*)
Bases: *katcp.sampling.SampleStrategy*

Event rate sampling strategy.

Report the sensor value whenever it changes or if more than *longest_period* seconds have passed since the last reported update. However, do not report the value if less than *shortest_period* seconds have passed since the last reported update.

Methods

SampleEventRate.attach()	Attach strategy to its sensor and send initial update.
SampleEventRate.cancel()	Detach strategy from its sensor and cancel ioloop callbacks.
SampleEventRate.cancel_timeouts()	Override this method to cancel any outstanding ioloop timeouts.
SampleEventRate.detach()	Detach strategy from its sensor.
SampleEventRate.get_sampling()	The Strategy constant for this sampling strategy.
SampleEventRate. get_sampling_formatted()	The current sampling strategy and parameters.
SampleEventRate. get_strategy(strategyName, ...)	Factory method to create a strategy object.
SampleEventRate.inform(reading)	Inform strategy creator of the sensor status.
SampleEventRate.start()	Start operating the strategy.
SampleEventRate.update(sensor, reading)	Callback used by the sensor's notify() method.

cancel_timeouts()
Override this method to cancel any outstanding ioloop timeouts.

get_sampling()
The Strategy constant for this sampling strategy.

Sub-classes should implement this method and return the appropriate constant.

Returns **strategy** : Strategy constant

The strategy type constant for this strategy.

inform(*reading*)
Inform strategy creator of the sensor status.

start()
Start operating the strategy.

Subclasses that override start() should call the super method before it does anything that uses the ioloop. This will attach to the sensor as an observer if `OBSERVE_UPDATES` is True, and sets `_ioloop_thread_id` using *thread.get_ident()*.

update(*sensor*, *reading*)
Callback used by the sensor's notify() method.

This update method is called whenever the sensor value is set so sensor will contain the right info. Note that the strategy does not really need to be passed a sensor because it already has a handle to it, but receives it due to the generic observer mechanism.

Sub-classes should override this method or *start()* to provide the necessary sampling strategy. Sub-classes should also ensure that *update()* is thread-safe; an easy way to do this is by using the @update_in_ioloop decorator.

Parameters **sensor** : Sensor object

The sensor which was just updated.

reading : (timestamp, status, value) tuple

Sensor reading as would be returned by sensor.read()

class katcp.sampling.**SampleNone**(*inform_callback*, *sensor*, **params*, ***kwargs*)
Bases: *katcp.sampling.SampleStrategy*

Sampling strategy which never sends any updates.

Methods

SampleNone.attach()	Attach strategy to its sensor and send initial update.
SampleNone.cancel()	Detach strategy from its sensor and cancel ioloop callbacks.
SampleNone.cancel_timeouts()	Override this method to cancel any outstanding ioloop timeouts.
SampleNone.detach()	Detach strategy from its sensor.
SampleNone.get_sampling()	The Strategy constant for this sampling strategy.
SampleNone.get_sampling_formatted()	The current sampling strategy and parameters.
SampleNone.get_strategy(strategyName, ...)	Factory method to create a strategy object.
SampleNone.inform(reading)	Inform strategy creator of the sensor status.
SampleNone.start()	Start operating the strategy.
SampleNone.update(sensor, reading)	Callback used by the sensor's notify() method.

get_sampling()
The Strategy constant for this sampling strategy.

Sub-classes should implement this method and return the appropriate constant.

Returns **strategy** : Strategy constant

The strategy type constant for this strategy.

start()
Start operating the strategy.

Subclasses that override start() should call the super method before it does anything that uses the ioloop. This will attach to the sensor as an observer if OBSERVE_UPDATES is True, and sets _ioloop_thread_id using *thread.get_ident()*.

class katcp.sampling.**SamplePeriod**(*inform_callback*, *sensor*, **params*, ***kwargs*)
Bases: *katcp.sampling.SampleStrategy*

Periodic sampling strategy.

Methods

SamplePeriod.attach()	Attach strategy to its sensor and send initial update.
SamplePeriod.cancel()	Detach strategy from its sensor and cancel ioloop callbacks.
SamplePeriod.cancel_timeouts()	Override this method to cancel any outstanding ioloop timeouts.
SamplePeriod.detach()	Detach strategy from its sensor.
SamplePeriod.get_sampling()	The Strategy constant for this sampling strategy.
SamplePeriod.get_sampling_formatted()	The current sampling strategy and parameters.
SamplePeriod.get_strategy(strategyName, ...)	Factory method to create a strategy object.
SamplePeriod.inform(reading)	Inform strategy creator of the sensor status.
SamplePeriod.start()	Start operating the strategy.
SamplePeriod.update(sensor, reading)	Callback used by the sensor's notify() method.

cancel_timeouts()
Override this method to cancel any outstanding ioloop timeouts.

get_sampling()
The Strategy constant for this sampling strategy.

Sub-classes should implement this method and return the appropriate constant.

Returns **strategy** : Strategy constant

The strategy type constant for this strategy.

start()
Start operating the strategy.

Subclasses that override start() should call the super method before it does anything that uses the ioloop. This will attach to the sensor as an observer if OBSERVE_UPDATES is True, and sets _ioloop_thread_id using *thread.get_ident()*.

class katcp.sampling.**SampleStrategy**(*inform_callback*, *sensor*, **params*, ***kwargs*)
Bases: object

Base class for strategies for sampling sensors.

Parameters **inform_callback** : callable, signature inform_callback(sensor_obj, reading)

Callback to receive inform messages.

sensor : Sensor object

Sensor to sample.

params : list of objects

Custom sampling parameters.

Methods

SampleStrategy.attach()	Attach strategy to its sensor and send initial update.
SampleStrategy.cancel()	Detach strategy from its sensor and cancel ioloop callbacks.
SampleStrategy.cancel_timeouts()	Override this method to cancel any outstanding ioloop timeouts.
SampleStrategy.detach()	Detach strategy from its sensor.
SampleStrategy.get_sampling()	The Strategy constant for this sampling strategy.
SampleStrategy.get_sampling_formatted()	The current sampling strategy and parameters.
SampleStrategy.get_strategy(strategyName, ...)	Factory method to create a strategy object.
SampleStrategy.inform(reading)	Inform strategy creator of the sensor status.
SampleStrategy.start()	Start operating the strategy.
SampleStrategy.update(sensor, reading)	Callback used by the sensor's notify() method.

OBSERVE_UPDATES = False
True if a strategy must be attached to its sensor as an observer

attach()
Attach strategy to its sensor and send initial update.

cancel()
Detach strategy from its sensor and cancel ioloop callbacks.

cancel_timeouts()
Override this method to cancel any outstanding ioloop timeouts.

detach()
Detach strategy from its sensor.

get_sampling()
The Strategy constant for this sampling strategy.

Sub-classes should implement this method and return the appropriate constant.

Returns **strategy** : Strategy constant

The strategy type constant for this strategy.

get_sampling_formatted()
The current sampling strategy and parameters.

The strategy is returned as a byte string and the values in the parameter list are formatted as byte strings using the formatter for this sensor type.

Returns **strategy_name** : bytes

KATCP name for the strategy.

params : list of bytes

KATCP formatted parameters for the strategy.

classmethod get_strategy(*strategyName*, *inform_callback*, *sensor*, **params*, ***kwargs*)

Factory method to create a strategy object.

Parameters **strategyName** : str or bytes

Name of strategy.

inform_callback : callable, signature inform_callback(sensor, reading)

Callback to receive inform messages.

sensor : Sensor object

Sensor to sample.

params : list of objects

Custom sampling parameters for specified strategy.

Keyword Arguments **ioloop** : tornado.ioloop.IOLoop instance, optional

Tornado ioloop to use, otherwise tornado.ioloop.IOLoop.current()

Returns **strategy** : *SampleStrategy* object

The created sampling strategy.

inform(*reading*)

Inform strategy creator of the sensor status.

start()

Start operating the strategy.

Subclasses that override start() should call the super method before it does anything that uses the ioloop. This will attach to the sensor as an observer if *OBSERVE_UPDATES* is True, and sets `_ioloop_thread_id` using *thread.get_ident()*.

update(*sensor*, *reading*)

Callback used by the sensor's notify() method.

This update method is called whenever the sensor value is set so sensor will contain the right info. Note that the strategy does not really need to be passed a sensor because it already has a handle to it, but receives it due to the generic observer mechanism.

Sub-classes should override this method or *start()* to provide the necessary sampling strategy. Sub-classes should also ensure that *update()* is thread-safe; an easy way to do this is by using the @update_in_ioloop decorator.

Parameters **sensor** : Sensor object

The sensor which was just updated.

reading : (timestamp, status, value) tuple

Sensor reading as would be returned by sensor.read()

katcp.sampling.**update_in_ioloop**(*update*)

Decorator that ensures an update() method is run in the tornado ioloop.

Does this by checking the thread identity. Requires that the object to which the method is bound has the attributes `_ioloop_thread_id` (the result of thread.get_ident() in the ioloop thread) and `ioloop` (the ioloop instance in use). Also assumes the signature *update(self, sensor, reading)* for the method.

1.9 KATCP Server API (server)

Servers for the KAT device control language.

class katcp.server.**AsyncDeviceServer**(**args*, ***kwargs*)

Bases: *katcp.server.DeviceServer*

DeviceServer that is automatically configured for async use.

Same as instantiating a *DeviceServer* instance and calling methods *set_concurrency_options(thread_safe=False, handler_thread=False)* and *set_ioloop(tornado.ioloop.IOLoop.current())* before starting.

Methods

AsyncDeviceServer.add_sensor(sensor)	Add a sensor to the device.
AsyncDeviceServer.build_state()	Return build state string of the form name-major.minor[(a|b|rc)n].
AsyncDeviceServer.clear_strategies(client_conn)	Clear the sensor strategies of a client connection.
AsyncDeviceServer.create_exception_reply_and_log(...)	
AsyncDeviceServer.create_log_inform(...[, ...])	Create a katcp logging inform message.
AsyncDeviceServer.get_sensor(sensor_name)	Fetch the sensor with the given name.
AsyncDeviceServer.get_sensors()	Fetch a list of all sensors.
AsyncDeviceServer.handle_inform(connection, msg)	Dispatch an inform message to the appropriate method.
AsyncDeviceServer.handle_message(...)	Handle messages of all types from clients.
AsyncDeviceServer.handle_reply(connection, msg)	Dispatch a reply message to the appropriate method.
AsyncDeviceServer.handle_request(connection, msg)	Dispatch a request message to the appropriate method.
AsyncDeviceServer.has_sensor(sensor_name)	Whether the sensor with specified name is known.
AsyncDeviceServer.inform(connection, msg)	Send an inform message to a particular client.
AsyncDeviceServer.join([timeout])	Rejoin the server thread.
AsyncDeviceServer.mass_inform(msg)	Send an inform message to all clients.
AsyncDeviceServer.next()	
AsyncDeviceServer.on_client_connect(**kwargs)	Inform client of build state and version on connect.
AsyncDeviceServer.on_client_disconnect(...)	Inform client it is about to be disconnected.

Continued on next page

Table 70 – continued from previous page

AsyncDeviceServer.on_message(client_conn, msg)	Dummy implementation of on_message required by KATCPServer.
AsyncDeviceServer.remove_sensor(sensor)	Remove a sensor from the device.
AsyncDeviceServer.reply(connection, reply, ...)	Send an asynchronous reply to an earlier request.
AsyncDeviceServer.reply_inform(connection, ...)	Send an inform as part of the reply to an earlier request.
AsyncDeviceServer.request_client_list(req, msg)	Request the list of connected clients.
AsyncDeviceServer.request_halt(req, msg)	Halt the device server.
AsyncDeviceServer.request_help(req, msg)	Return help on the available requests.
AsyncDeviceServer.request_log_level(req, msg)	Query or set the current logging level.
AsyncDeviceServer.request_request_timeout_hint(...)	Return timeout hints for requests
AsyncDeviceServer.request_restart(req, msg)	Restart the device server.
AsyncDeviceServer.request_sensor_list(req, msg)	Request the list of sensors.
AsyncDeviceServer.request_sensor_sampling(...)	Configure or query the way a sensor is sampled.
AsyncDeviceServer.request_sensor_sampling_clear(...)	Set all sampling strategies for this client to none.
AsyncDeviceServer.request_sensor_value(req, msg)	Request the value of a sensor or sensors.
AsyncDeviceServer.request_version_list(req, msg)	Request the list of versions of roles and subcomponents.
AsyncDeviceServer.request_watchdog(req, msg)	Check that the server is still alive.
AsyncDeviceServer.running()	Whether the server is running.
AsyncDeviceServer.setDaemon(daemonic)	Set daemonic state of the managed ioloop thread to True / False
AsyncDeviceServer.set_concurrency_options([...])	Set concurrency options for this device server.
AsyncDeviceServer.set_ioloop([ioloop])	Set the tornado IOLoop to use.
AsyncDeviceServer.set_restart_queue(...)	Set the restart queue.
AsyncDeviceServer.setup_sensors()	Populate the dictionary of sensors.
AsyncDeviceServer.start([timeout])	Start the server in a new thread.
AsyncDeviceServer.stop([timeout])	Stop a running server (from another thread).
AsyncDeviceServer.sync_with_ioloop([timeout])	Block for ioloop to complete a loop if called from another thread.
AsyncDeviceServer.version()	Return a version string of the form type-major.minor.
AsyncDeviceServer.wait_running([timeout])	Wait until the server is running

katcp.server.**BASE_REQUESTS = frozenset(['sensor-sampling', 'help', 'new-command', 'raise-fa**

List of basic KATCP requests that a minimal device server should implement

class katcp.server.**ClientConnection**(*server*, *conn_id*)
Bases: future.types.newobject.newobject

Encapsulates the connection between a single client and the server.

Methods

ClientConnection.disconnect(reason)	Disconnect this client connection for specified reason
ClientConnection.inform(msg)	Send an inform message to a particular client.
ClientConnection.mass_inform(msg)	Send an inform message to all clients.
ClientConnection.next()	
ClientConnection.on_client_disconnect_was_called()	Prevent multiple calls to on_client_disconnect handler.
ClientConnection.reply(reply, orig_req)	Send an asynchronous reply to an earlier request.
ClientConnection.reply_inform(inform, orig_req)	Send an inform as part of the reply to an earlier request.

disconnect(*reason*)
Disconnect this client connection for specified reason

inform(*msg*)
Send an inform message to a particular client.

Should only be used for asynchronous informs. Informs that are part of the response to a request should use *reply_inform()* so that the message identifier from the original request can be attached to the inform.

Parameters **msg** : Message object

The inform message to send.

mass_inform(*msg*)
Send an inform message to all clients.

Parameters **msg** : Message object

The inform message to send.

on_client_disconnect_was_called()
Prevent multiple calls to on_client_disconnect handler.

Call this when an on_client_disconnect handler has been called.

reply(*reply*, *orig_req*)
Send an asynchronous reply to an earlier request.

Parameters **reply** : Message object

The reply message to send.

orig_req : Message object

The request message being replied to. The reply message's id is overridden with the id from orig_req before the reply is sent.

reply_inform(*inform*, *orig_req*)
Send an inform as part of the reply to an earlier request.

Parameters **inform** : Message object

The inform message to send.

orig_req : Message object

The request message being replied to. The inform message's id is overridden with the id from orig_req before the inform is sent.

class katcp.server.**ClientRequestConnection**(*client_connection*, *req_msg*)
Bases: future.types.newobject.newobject

Encapsulates specific KATCP request and associated client connection.

Methods

ClientRequestConnection.inform(*args)	
ClientRequestConnection.inform_after_reply(*args)	
ClientRequestConnection.make_reply(*args)	
ClientRequestConnection.next()	
ClientRequestConnection.reply(*args)	
ClientRequestConnection.reply_again(*args)	
ClientRequestConnection.reply_with_message(rep_msg)	Send a pre-created reply message to the client connection.

reply_with_message(*rep_msg*)
Send a pre-created reply message to the client connection.

Will check that rep_msg.name matches the bound request.

class katcp.server.**DeviceLogger**(*device_server*, *root_logger='root'*, *python_logger=None*)
Bases: future.types.newobject.newobject

Object for logging messages from a DeviceServer.

Log messages are logged at a particular level and under a particular name. Names use dotted notation to form a virtual hierarchy of loggers with the device.

Parameters **device_server** : DeviceServerBase object

The device server this logger should use for sending out logs.

root_logger : str

The name of the root logger.

Methods

DeviceLogger.debug(msg, *args, **kwargs)	Log a debug message.
DeviceLogger.error(msg, *args, **kwargs)	Log an error message.
DeviceLogger.fatal(msg, *args, **kwargs)	Log a fatal error message.
DeviceLogger.info(msg, *args, **kwargs)	Log an info message.
DeviceLogger.level_from_name(level_name)	Return the level constant for a given name.
DeviceLogger.level_name([level])	Return the name of the given level value.

Continued on next page

Table 73 – continued from previous page

DeviceLogger.log(level, msg, *args, **kwargs)	Log a message and inform all clients.
DeviceLogger.log_to_python(logger, msg)	Log a KATCP logging message to a Python logger.
DeviceLogger.next()	
DeviceLogger.set_log_level(level)	Set the logging level.
DeviceLogger.set_log_level_by_name(level_name)	Set the logging level using a level name.
DeviceLogger.trace(msg, *args, **kwargs)	Log a trace message.
DeviceLogger.warn(msg, *args, **kwargs)	Log an warning message.

debug(*msg*, **args*, ***kwargs*)
Log a debug message.

error(*msg*, **args*, ***kwargs*)
Log an error message.

fatal(*msg*, **args*, ***kwargs*)
Log a fatal error message.

info(*msg*, **args*, ***kwargs*)
Log an info message.

level_from_name(*level_name*)
Return the level constant for a given name.

If the *level_name* is not known, raise a ValueError.

Parameters **level_name** : str or bytes

The logging level name whose logging level constant to retrieve.

Returns **level** : logging level constant

The logging level constant associated with the name.

level_name(*level=None*)
Return the name of the given level value.

If level is None, return the name of the current level.

Parameters **level** : logging level constant

The logging level constant whose name to retrieve.

Returns **level_name** : str

The name of the logging level.

log(*level*, *msg*, **args*, ***kwargs*)
Log a message and inform all clients.

Parameters **level** : logging level constant

The level to log the message at.

msg : str

The text format for the log message.

args : list of objects

Arguments to pass to log format string. Final message text is created using: msg % args.

kwargs : additional keyword parameters

Allowed keywords are 'name' and 'timestamp'. The name is the name of the logger to log the message to. If not given the name defaults to the root logger. The timestamp is a float in seconds. If not given the timestamp defaults to the current time.

classmethod log_to_python(*logger*, *msg*)
Log a KATCP logging message to a Python logger.

Parameters **logger** : logging.Logger object

The Python logger to log the given message to.

msg : Message object

The #log message to create a log entry from.

set_log_level(*level*)
Set the logging level.

Parameters **level** : logging level constant

The value to set the logging level to.

set_log_level_by_name(*level_name*)
Set the logging level using a level name.

Parameters **level_name** : str or bytes

The name of the logging level.

trace(*msg*, **args*, ***kwargs*)
Log a trace message.

warn(*msg*, **args*, ***kwargs*)
Log an warning message.

class katcp.server.**DeviceServer**(**args*, ***kwargs*)
Bases: *katcp.server.DeviceServerBase*

Implements some standard messages on top of DeviceServerBase.

Inform messages handled are:

- version (sent on connect)
- build-state (sent on connect)
- log (via self.log.warn(...), etc)
- disconnect
- client-connected

Requests handled are:

- halt
- help
- log-level
- restart[1]
- client-list
- sensor-list

[1] Restart relies on .set_restart_queue() being used to register a restart queue with the device. When the device needs to be restarted, it will be added to the restart queue. The queue should be a Python Queue.Queue object without a maximum size.

- sensor-sampling
- sensor-value
- watchdog
- version-list (only standard in KATCP v5 or later)
- **request-timeout-hint (pre-standard only if protocol flags indicates** timeout hints, supported for KATCP v5.1 or later)
- sensor-sampling-clear (non-standard)

Unhandled standard requests are:

- configure
- mode

Subclasses can define the tuple VERSION_INFO to set the interface name, major and minor version numbers. The BUILD_INFO tuple can be defined to give a string describing a particular interface instance and may have a fourth element containing additional version information (e.g. rc1).

Subclasses may manipulate the versions returned by the ?version-list command by editing .extra_versions which is a dictionary mapping role or component names to (version, build_state_or_serial_no) tuples. The build_state_or_serial_no may be None.

Subclasses must override the .setup_sensors() method. If they have no sensors to register, the method should just be a pass.

Methods

DeviceServer.add_sensor(sensor)	Add a sensor to the device.
DeviceServer.build_state()	Return build state string of the form name-major.minor[(a|b|rc)n].
DeviceServer.clear_strategies(client_conn[, ...])	Clear the sensor strategies of a client connection.
DeviceServer.create_exception_reply_and_log(...)	
DeviceServer.create_log_inform(level_name, ...)	Create a katcp logging inform message.
DeviceServer.get_sensor(sensor_name)	Fetch the sensor with the given name.
DeviceServer.get_sensors()	Fetch a list of all sensors.
DeviceServer.handle_inform(connection, msg)	Dispatch an inform message to the appropriate method.
DeviceServer.handle_message(client_conn, msg)	Handle messages of all types from clients.
DeviceServer.handle_reply(connection, msg)	Dispatch a reply message to the appropriate method.
DeviceServer.handle_request(connection, msg)	Dispatch a request message to the appropriate method.
DeviceServer.has_sensor(sensor_name)	Whether the sensor with specified name is known.
DeviceServer.inform(connection, msg)	Send an inform message to a particular client.
DeviceServer.join([timeout])	Rejoin the server thread.
DeviceServer.mass_inform(msg)	Send an inform message to all clients.
DeviceServer.next()	
DeviceServer.on_client_connect(**kwargs)	Inform client of build state and version on connect.

Continued on next page

Table 74 – continued from previous page

DeviceServer.on_client_disconnect(...)	Inform client it is about to be disconnected.
DeviceServer.on_message(client_conn, msg)	Dummy implementation of on_message required by KATCPServer.
DeviceServer.remove_sensor(sensor)	Remove a sensor from the device.
DeviceServer.reply(connection, reply, orig_req)	Send an asynchronous reply to an earlier request.
DeviceServer.reply_inform(connection, ...)	Send an inform as part of the reply to an earlier request.
DeviceServer.request_client_list(req, msg)	Request the list of connected clients.
DeviceServer.request_halt(req, msg)	Halt the device server.
DeviceServer.request_help(req, msg)	Return help on the available requests.
DeviceServer.request_log_level(req, msg)	Query or set the current logging level.
DeviceServer.request_request_timeout_hint(req, msg)	Return timeout hints for requests
DeviceServer.request_restart(req, msg)	Restart the device server.
DeviceServer.request_sensor_list(req, msg)	Request the list of sensors.
DeviceServer.request_sensor_sampling(req, msg)	Configure or query the way a sensor is sampled.
DeviceServer.request_sensor_sampling_clear(req)	Set all sampling strategies for this client to none.
DeviceServer.request_sensor_value(req, msg)	Request the value of a sensor or sensors.
DeviceServer.request_version_list(req, msg)	Request the list of versions of roles and subcomponents.
DeviceServer.request_watchdog(req, msg)	Check that the server is still alive.
DeviceServer.running()	Whether the server is running.
DeviceServer.setDaemon(daemonic)	Set daemonic state of the managed ioloop thread to True / False
DeviceServer.set_concurrency_options([...])	Set concurrency options for this device server.
DeviceServer.set_ioloop([ioloop])	Set the tornado IOLoop to use.
DeviceServer.set_restart_queue(restart_queue)	Set the restart queue.
DeviceServer.setup_sensors()	Populate the dictionary of sensors.
DeviceServer.start([timeout])	Start the server in a new thread.
DeviceServer.stop([timeout])	Stop a running server (from another thread).
DeviceServer.sync_with_ioloop([timeout])	Block for ioloop to complete a loop if called from another thread.
DeviceServer.version()	Return a version string of the form type-major.minor.
DeviceServer.wait_running([timeout])	Wait until the server is running

add_sensor(*sensor*)

Add a sensor to the device.

Usually called inside .setup_sensors() but may be called from elsewhere.

Parameters **sensor** : Sensor object

The sensor object to register with the device server.

build_state()

Return build state string of the form name-major.minor[(a|b|rc)n].

clear_strategies(*client_conn*, *remove_client=False*)

Clear the sensor strategies of a client connection.

Parameters **client_connection** : ClientConnection instance

The connection that should have its sampling strategies cleared

remove_client : bool, optional

Remove the client connection from the strategies data-structure. Useful for clients that disconnect.

get_sensor(*sensor_name*)
Fetch the sensor with the given name.

Parameters **sensor_name** : str

Name of the sensor to retrieve.

Returns **sensor** : Sensor object

The sensor with the given name.

get_sensors()
Fetch a list of all sensors.

Returns **sensors** : list of Sensor objects

The list of sensors registered with the device server.

has_sensor(*sensor_name*)
Whether the sensor with specified name is known.

on_client_connect(***kwargs*)
Inform client of build state and version on connect.

Parameters **client_conn** : ClientConnection object

The client connection that has been successfully established.

Returns **Future that resolves when the device is ready to accept messages.** :

on_client_disconnect(*client_conn*, *msg*, *connection_valid*)
Inform client it is about to be disconnected.

Parameters **client_conn** : ClientConnection object

The client connection being disconnected.

msg : str

Reason client is being disconnected.

connection_valid : bool

True if connection is still open for sending, False otherwise.

Returns **Future that resolves when the client connection can be closed.** :

remove_sensor(*sensor*)
Remove a sensor from the device.

Also deregisters all clients observing the sensor.

Parameters **sensor** : Sensor object or name string

The sensor to remove from the device server.

request_client_list (*req*, *msg*)
Request the list of connected clients.

The list of clients is sent as a sequence of #client-list informs.

Informs **addr** : str

The address of the client as host:port with host in dotted quad notation. If the address of the client could not be determined (because, for example, the client disconnected suddenly) then a unique string representing the client is sent instead.

Returns **success** : {'ok', 'fail'}

Whether sending the client list succeeded.

informs : int

Number of #client-list inform messages sent.

Examples

```
?client-list
#client-list 127.0.0.1:53600
!client-list ok 1
```

request_halt (*req*, *msg*)
Halt the device server.

Returns **success** : {'ok', 'fail'}

Whether scheduling the halt succeeded.

Examples

```
?halt
!halt ok
```

request_help (*req*, *msg*)
Return help on the available requests.

Return a description of the available requests using a sequence of #help informs.

Parameters **request** : str, optional

The name of the request to return help for (the default is to return help for all requests).

Informs **request** : str

The name of a request.

description : str

Documentation for the named request.

Returns **success** : {'ok', 'fail'}

Whether sending the help succeeded.

informs : int

Number of #help inform messages sent.

Examples

```
?help
#help halt ...description...
#help help ...description...
...
!help ok 5

?help halt
#help halt ...description...
!help ok 1
```

request_log_level (*req*, *msg*)
Query or set the current logging level.

Parameters **level** : {'all', 'trace', 'debug', 'info', 'warn', 'error', 'fatal', 'off'}, optional

Name of the logging level to set the device server to (the default is to leave the log level unchanged).

Returns **success** : {'ok', 'fail'}

Whether the request succeeded.

level : {'all', 'trace', 'debug', 'info', 'warn', 'error', 'fatal', 'off'}

The log level after processing the request.

Examples

```
?log-level
!log-level ok warn

?log-level info
!log-level ok info
```

request_request_timeout_hint (*req*, *msg*)
Return timeout hints for requests

KATCP requests should generally take less than 5s to complete, but some requests are unavoidably slow. This results in spurious client timeout errors. This request provides timeout hints that clients can use to select suitable request timeouts.

Parameters **request** : str, optional

The name of the request to return a timeout hint for (the default is to return hints for all requests that have timeout hints). Returns one inform per request. Must be an existing request if specified.

Informs **request** : str

The name of the request.

suggested_timeout : float

Suggested request timeout in seconds for the request. If *suggested_timeout* is zero (0), no timeout hint is available.

Returns **success** : {'ok', 'fail'}

Whether sending the help succeeded.

informs : int

Number of #request-timeout-hint inform messages sent.

Notes

?request-timeout-hint without a parameter will only return informs for requests that have specific timeout hints, so it will most probably be a subset of all the requests, or even no informs at all.

Examples

```
?request-timeout-hint
#request-timeout-hint halt 5
#request-timeout-hint very-slow-request 500
...
!request-timeout-hint ok 5

?request-timeout-hint moderately-slow-request
#request-timeout-hint moderately-slow-request 20
!request-timeout-hint ok 1
```

request_restart(*req*, *msg*)

Restart the device server.

Returns **success** : {'ok', 'fail'}

Whether scheduling the restart succeeded.

Examples

```
?restart
!restart ok
```

request_sensor_list(*req*, *msg*)

Request the list of sensors.

The list of sensors is sent as a sequence of #sensor-list informs.

Parameters **name** : str, optional

Name of the sensor to list (the default is to list all sensors). If name starts and ends with '/' it is treated as a regular expression and all sensors whose names contain the regular expression are returned.

Informs **name** : str

The name of the sensor being described.

description : str

Description of the named sensor.

units : str

Units for the value of the named sensor.

type : str

Type of the named sensor.

params : list of str, optional

Additional sensor parameters (type dependent). For integer and float sensors the additional parameters are the minimum and maximum sensor value. For discrete sensors the additional parameters are the allowed values. For all other types no additional parameters are sent.

Returns **success** : {'ok', 'fail'}

Whether sending the sensor list succeeded.

informs : int

Number of #sensor-list inform messages sent.

Examples

```
?sensor-list
#sensor-list psu.voltage PSU\_voltage. V float 0.0 5.0
#sensor-list cpu.status CPU\_status. \@ discrete on off error
...
!sensor-list ok 5

?sensor-list cpu.power.on
#sensor-list cpu.power.on Whether\_CPU\_hase\_power. \@ boolean
!sensor-list ok 1

?sensor-list /voltage/
#sensor-list psu.voltage PSU\_voltage. V float 0.0 5.0
#sensor-list cpu.voltage CPU\_voltage. V float 0.0 3.0
!sensor-list ok 2
```

request_sensor_sampling(*req*, *msg*)

Configure or query the way a sensor is sampled.

Sampled values are reported asynchronously using the #sensor-status message.

Parameters **names** : str

One or more names of sensors whose sampling strategy will be queried or configured. If specifying multiple sensors, these must be provided as a comma-separated list. A query can only be done on a single sensor. However, configuration can be done on many sensors with a single request, as long as they all use the same strategy. Note: prior to KATCP v5.1 only a single sensor could be configured. Multiple sensors are only allowed if the device server sets the protocol version to KATCP v5.1 or higher and enables the BULK_SET_SENSOR_SAMPLING flag in its PROTOCOL_INFO class attribute.

strategy : {'none', 'auto', 'event', 'differential', 'differential-rate',

'period', 'event-rate'}, optional

Type of strategy to use to report the sensor value. The differential strategy types may only be used with integer or float sensors. If this parameter is supplied, it sets the new strategy.

params : list of str, optional

Additional strategy parameters (dependent on the strategy type). For the differential strategy, the parameter is an integer or float giving the amount by which the sensor value may change before an updated value is sent. For the period strategy, the parameter is the

sampling period in float seconds. The event strategy has no parameters. Note that this has changed from KATCPv4. For the event-rate strategy, a minimum period between updates and a maximum period between updates (both in float seconds) must be given. If the event occurs more than once within the minimum period, only one update will occur. Whether or not the event occurs, the sensor value will be updated at least once per maximum period. For the differential-rate strategy there are 3 parameters. The first is the same as the differential strategy parameter. The second and third are the minimum and maximum periods, respectively, as with the event-rate strategy.

Informs timestamp : float

Timestamp of the sensor reading in seconds since the Unix epoch, or milliseconds for katcp versions <= 4.

count : {1}

Number of sensors described in this #sensor-status inform. Will always be one. It exists to keep this inform compatible with #sensor-value.

name : str

Name of the sensor whose value is being reported.

value : object

Value of the named sensor. Type depends on the type of the sensor.

Returns success : {'ok', 'fail'}

Whether the sensor-sampling request succeeded.

names : str

Name(s) of the sensor queried or configured. If multiple sensors, this will be a comma-separated list.

strategy : {'none', 'auto', 'event', 'differential', 'differential-rate',

'period', 'event-rate'}.

Name of the new or current sampling strategy for the sensor(s).

params : list of str

Additional strategy parameters (see description under Parameters).

Examples :

——— :

:: :

?sensor-sampling cpu.power.on !sensor-sampling ok cpu.power.on none

?sensor-sampling cpu.power.on period 0.5 #sensor-status 1244631611.415231 1 cpu.power.on nominal 1 !sensor-sampling ok cpu.power.on period 0.5

if BULK_SET_SENSOR_SAMPLING is enabled then:

?sensor-sampling cpu.power.on,fan.speed !sensor-sampling fail Cannot_query_multiple_sensors

?sensor-sampling cpu.power.on,fan.speed period 0.5 #sensor-status 1244631611.415231 1 cpu.power.on nominal 1 #sensor-status 1244631611.415200 1 fan.speed nominal 10.0 !sensor-sampling ok cpu.power.on,fan.speed period 0.5

request_sensor_sampling_clear (*req*, *msg*)
Set all sampling strategies for this client to none.

Returns **success** : {'ok', 'fail'}

Whether sending the list of devices succeeded.

Examples

?sensor-sampling-clear !sensor-sampling-clear ok

request_sensor_value (*req*, *msg*)
Request the value of a sensor or sensors.

A list of sensor values as a sequence of #sensor-value informs.

Parameters **name** : str, optional

Name of the sensor to poll (the default is to send values for all sensors). If name starts and ends with '/' it is treated as a regular expression and all sensors whose names contain the regular expression are returned.

Informs **timestamp** : float

Timestamp of the sensor reading in seconds since the Unix epoch, or milliseconds for katcp versions <= 4.

count : {1}

Number of sensors described in this #sensor-value inform. Will always be one. It exists to keep this inform compatible with #sensor-status.

name : str

Name of the sensor whose value is being reported.

value : object

Value of the named sensor. Type depends on the type of the sensor.

Returns **success** : {'ok', 'fail'}

Whether sending the list of values succeeded.

informs : int

Number of #sensor-value inform messages sent.

Examples

```
?sensor-value
#sensor-value 1244631611.415231 1 psu.voltage nominal 4.5
#sensor-value 1244631611.415200 1 cpu.status nominal off
...
!sensor-value ok 5

?sensor-value cpu.power.on
#sensor-value 1244631611.415231 1 cpu.power.on nominal 0
!sensor-value ok 1
```

request_version_list (*req*, *msg*)
Request the list of versions of roles and subcomponents.

Informs **name** : str

Name of the role or component.

version : str

A string identifying the version of the component. Individual components may define the structure of this argument as they choose. In the absence of other information clients should treat it as an opaque string.

build_state_or_serial_number : str

A unique identifier for a particular instance of a component. This should change whenever the component is replaced or updated.

Returns **success** : {'ok', 'fail'}

Whether sending the version list succeeded.

informs : int

Number of #version-list inform messages sent.

Examples

```
?version-list
#version-list katcp-protocol 5.0-MI
#version-list katcp-library katcp-python-0.4 katcp-python-0.4.1-py2
#version-list katcp-device foodevice-1.0 foodevice-1.0.0rc1
!version-list ok 3
```

request_watchdog(*req*, *msg*)

Check that the server is still alive.

Returns **success** : {'ok'}

Examples

```
?watchdog
!watchdog ok
```

set_restart_queue(*restart_queue*)

Set the restart queue.

When the device server should be restarted, it will be added to the queue.

Parameters **restart_queue** : Queue.Queue object

The queue to add the device server to when it should be restarted.

setup_sensors()

Populate the dictionary of sensors.

Unimplemented by default – subclasses should add their sensors here or pass if there are no sensors.

Examples

```
>>> class MyDevice(DeviceServer):
...     def setup_sensors(self):
...         self.add_sensor(Sensor(...))
...         self.add_sensor(Sensor(...))
...
```

version()
Return a version string of the form type-major.minor.

class katcp.server.**DeviceServerBase**(*host*, *port*, *tb_limit=20*, *logger=<logging.Logger object>*)

Bases: future.types.newobject.newobject

Base class for device servers.

Subclasses should add .request_* methods for dealing with request messages. These methods each take the client request connection and msg objects as arguments and should return the reply message or raise an exception as a result.

Subclasses can also add .inform_* and reply_* methods to handle those types of messages.

Should a subclass need to generate inform messages it should do so using either the .inform() or .mass_inform() methods.

Finally, this class should probably not be subclassed directly but rather via subclassing DeviceServer itself which implements common .request_* methods.

Parameters **host** : str

Host to listen on.

port : int

Port to listen on.

tb_limit : int, optional

Maximum number of stack frames to send in error tracebacks.

logger : logging.Logger object, optional

Logger to log messages to.

Methods

DeviceServerBase.create_exception_reply_and_log(...)	
DeviceServerBase.create_log_inform(...[, ...])	Create a katcp logging inform message.
DeviceServerBase.handle_inform(connection, msg)	Dispatch an inform message to the appropriate method.
DeviceServerBase.handle_message(client_conn, msg)	Handle messages of all types from clients.
DeviceServerBase.handle_reply(connection, msg)	Dispatch a reply message to the appropriate method.
DeviceServerBase.handle_request(connection, msg)	Dispatch a request message to the appropriate method.

Continued on next page

Table 75 – continued from previous page

DeviceServerBase.inform(connection, msg)	Send an inform message to a particular client.
DeviceServerBase.join([timeout])	Rejoin the server thread.
DeviceServerBase.mass_inform(msg)	Send an inform message to all clients.
DeviceServerBase.next()	
DeviceServerBase.on_client_connect(**kwargs)	Called after client connection is established.
DeviceServerBase.on_client_disconnect(**kwargs)	Called before a client connection is closed.
DeviceServerBase.on_message(client_conn, msg)	Dummy implementation of on_message required by KATCPServer.
DeviceServerBase.reply(connection, reply, ...)	Send an asynchronous reply to an earlier request.
DeviceServerBase.reply_inform(connection, ...)	Send an inform as part of the reply to an earlier request.
DeviceServerBase.running()	Whether the server is running.
DeviceServerBase.setDaemon(daemonic)	Set daemonic state of the managed ioloop thread to True / False
DeviceServerBase.set_concurrency_options([...])	Set concurrency options for this device server.
DeviceServerBase.set_ioloop([ioloop])	Set the tornado IOLoop to use.
DeviceServerBase.start([timeout])	Start the server in a new thread.
DeviceServerBase.stop([timeout])	Stop a running server (from another thread).
DeviceServerBase.sync_with_ioloop([timeout])	Block for ioloop to complete a loop if called from another thread.
DeviceServerBase.wait_running([timeout])	Wait until the server is running

create_log_inform(*level_name*, *msg*, *name*, *timestamp=None*)
Create a katcp logging inform message.

Usually this will be called from inside a DeviceLogger object, but it is also used by the methods in this class when errors need to be reported to the client.

handle_inform(*connection*, *msg*)
Dispatch an inform message to the appropriate method.

Parameters **connection** : ClientConnection object

The client connection the message was from.

msg : Message object

The inform message to process.

handle_message(*client_conn*, *msg*)
Handle messages of all types from clients.

Parameters **client_conn** : ClientConnection object

The client connection the message was from.

msg : Message object

The message to process.

handle_reply(*connection*, *msg*)
Dispatch a reply message to the appropriate method.

Parameters **connection** : ClientConnection object

The client connection the message was from.

msg : Message object

The reply message to process.

handle_request(*connection*, *msg*)
Dispatch a request message to the appropriate method.

Parameters **connection** : ClientConnection object

The client connection the message was from.

msg : Message object

The request message to process.

Returns **done_future** : Future or None

Returns Future for async request handlers that will resolve when done, or None for sync request handlers once they have completed.

inform(*connection*, *msg*)
Send an inform message to a particular client.

Should only be used for asynchronous informs. Informs that are part of the response to a request should use *reply_inform()* so that the message identifier from the original request can be attached to the inform.

Parameters **connection** : ClientConnection object

The client to send the message to.

msg : Message object

The inform message to send.

join(*timeout=None*)
Rejoin the server thread.

Parameters **timeout** : float or None, optional

Time in seconds to wait for the thread to finish.

mass_inform(*msg*)
Send an inform message to all clients.

Parameters **msg** : Message object

The inform message to send.

on_client_connect(***kwargs*)
Called after client connection is established.

Subclasses should override if they wish to send clients message or perform house-keeping at this point.

Parameters **conn** : ClientConnection object

The client connection that has been successfully established.

Returns **Future that resolves when the device is ready to accept messages.** :

on_client_disconnect(***kwargs*)
Called before a client connection is closed.

Subclasses should override if they wish to send clients message or perform house-keeping at this point. The server cannot guarantee this will be called (for example, the client might drop the connection). The message parameter contains the reason for the disconnection.

Parameters **conn** : ClientConnection object

Client connection being disconnected.

msg : str

Reason client is being disconnected.

connection_valid : boolean

True if connection is still open for sending, False otherwise.

Returns **Future that resolves when the client connection can be closed.** :

on_message(*client_conn*, *msg*)

Dummy implementation of on_message required by KATCPServer.

Will be replaced by a handler with the appropriate concurrency semantics when set_concurrency_options is called (defaults are set in __init__()).

reply(*connection*, *reply*, *orig_req*)

Send an asynchronous reply to an earlier request.

Parameters **connection** : ClientConnection object

The client to send the reply to.

reply : Message object

The reply message to send.

orig_req : Message object

The request message being replied to. The reply message's id is overridden with the id from orig_req before the reply is sent.

reply_inform(*connection*, *inform*, *orig_req*)

Send an inform as part of the reply to an earlier request.

Parameters **connection** : ClientConnection object

The client to send the inform to.

inform : Message object

The inform message to send.

orig_req : Message object

The request message being replied to. The inform message's id is overridden with the id from orig_req before the inform is sent.

running()

Whether the server is running.

setDaemon(*daemonic*)

Set daemonic state of the managed ioloop thread to True / False

Calling this method for a non-managed ioloop has no effect. Must be called before start(), or it will also have no effect

set_concurrency_options(*thread_safe=True*, *handler_thread=True*)

Set concurrency options for this device server. Must be called before *start()*.

Parameters **thread_safe** : bool

If True, make the server public methods thread safe. Incurs performance overhead.

handler_thread : bool

Can only be set if *thread_safe* is True. Handle all requests (even from different clients) in a separate, single, request-handling thread. Blocking request handlers will prevent the server from handling new requests from any client, but sensor strategies should still function. This more or less mimics the behaviour of a server in library versions before 0.6.0.

set_ioloop(*ioloop=None*)

Set the tornado IOLoop to use.

Sets the tornado.ioloop.IOLoop instance to use, defaulting to IOLoop.current(). If set_ioloop() is never called the IOLoop is started in a new thread, and will be stopped if self.stop() is called.

Notes

Must be called before start() is called.

start(*timeout=None*)

Start the server in a new thread.

Parameters **timeout** : float or None, optional

Time in seconds to wait for server thread to start.

stop(*timeout=1.0*)

Stop a running server (from another thread).

Parameters **timeout** : float, optional

Seconds to wait for server to have *started*.

Returns **stopped** : thread-safe Future

Resolves when the server is stopped

sync_with_ioloop(*timeout=None*)

Block for ioloop to complete a loop if called from another thread.

Returns a future if called from inside the ioloop.

Raises concurrent.futures.TimeoutError if timed out while blocking.

wait_running(*timeout=None*)

Wait until the server is running

class katcp.server.**KATCPServer**(*device*, *host*, *port*, *tb_limit=20*, *logger=<logging.Logger object>*)

Bases: future.types.newobject.newobject

Tornado IO backend for a KATCP Device.

Listens for connections on a server socket, reads KATCP messages off the wire and passes them on to a DeviceServer-like class.

All class CONSTANT attributes can be changed until start() is called.

Methods

KATCPServer.call_from_thread(fn)	Allow thread-safe calls to ioloop functions.
KATCPServer.client_connection_factory	Factory that produces a ClientConnection compatible instance.
KATCPServer.flush_on_close(stream)	Flush tornado iostream write buffer and prevent further writes.
KATCPServer.get_address(stream)	Text representation of the network address of a connection stream.
KATCPServer.in_ioloop_thread()	Return True if called in the IOLoop thread of this server.
KATCPServer.join([timeout])	Rejoin the server thread.
KATCPServer.mass_send_message(msg)	Send a message to all connected clients.
KATCPServer.mass_send_message_from_thread(msg)	Thread-safe version of send_message() returning a Future instance.
KATCPServer.next()	
KATCPServer.running()	Whether the handler thread is running.
KATCPServer.send_message(stream, msg)	Send an arbitrary message to a particular client.
KATCPServer.send_message_from_thread(stream, msg)	Thread-safe version of send_message() returning a Future instance.
KATCPServer.setDaemon(daemonic)	Set daemonic state of the managed ioloop thread to True / False
KATCPServer.set_ioloop([ioloop])	Set the tornado IOLoop to use.
KATCPServer.start([timeout])	Install the server on its IOLoop, optionally starting the IOLoop.
KATCPServer.stop([timeout])	Stop a running server (from another thread).
KATCPServer.wait_running([timeout])	Wait until the handler thread is running.

DISCONNECT_TIMEOUT = 1

How long to wait for the device on_client_disconnect() to complete.

Note that this will only work if the device on_client_disconnect() method is non-blocking (i.e. returns a future immediately). Otherwise the ioloop will be blocked and unable to apply the timeout.

MAX_MSG_SIZE = 2097152

Maximum message size that can be received in bytes.

If more than MAX_MSG_SIZE bytes are read from the client without encountering a message terminator (i.e. newline), the connection is closed.

MAX_WRITE_BUFFER_SIZE = 4194304

Maximum outstanding bytes to be buffered by the server process.

If more than MAX_WRITE_BUFFER_SIZE bytes are outstanding, the client connection is closed. Note that the OS also buffers socket writes, so more than MAX_WRITE_BUFFER_SIZE bytes may be untransmitted in total.

bind_address

The (host, port) where the server is listening for connections.

call_from_thread(*fn*)

Allow thread-safe calls to ioloop functions.

Uses add_callback if not in the IOLoop thread, otherwise calls directly. Returns an already resolved *tornado.concurrent.Future* if in ioloop, otherwise a *concurrent.Future*. Logs unhandled exceptions. Resolves with an exception if one occurred.

client_connection_factory
Factory that produces a ClientConnection compatible instance.

signature: client_connection_factory(server, conn_id)

Should be set before calling start().

Methods

ClientConnection.disconnect(reason)	Disconnect this client connection for specified reason
ClientConnection.inform(msg)	Send an inform message to a particular client.
ClientConnection.mass_inform(msg)	Send an inform message to all clients.
ClientConnection.next()	
ClientConnection.on_client_disconnect_was_called()	Prevent multiple calls to on_client_disconnect handler.
ClientConnection.reply(reply, orig_req)	Send an asynchronous reply to an earlier request.
ClientConnection.reply_inform(inform, orig_req)	Send an inform as part of the reply to an earlier request.

alias of *ClientConnection*

flush_on_close(*stream*)
Flush tornado iostream write buffer and prevent further writes.

Returns a future that resolves when the stream is flushed.

get_address(*stream*)
Text representation of the network address of a connection stream.

Notes

This method is thread-safe

in_ioloop_thread()
Return True if called in the IOLoop thread of this server.

ioloop = None
The Tornado IOloop to use, set by self.set_ioloop()

join(*timeout=None*)
Rejoin the server thread.

Parameters **timeout** : float or None, optional

Time in seconds to wait for the thread to finish.

Notes

If the ioloop is not managed, this function will block until the server port is closed, meaning a new server can be started on the same port.

mass_send_message(*msg*)
Send a message to all connected clients.

Notes

This method can only be called in the IOLoop thread.

mass_send_message_from_thread(*msg*)
Thread-safe version of send_message() returning a Future instance.

See return value and notes for send_message_from_thread().

running()
Whether the handler thread is running.

send_message(*stream*, *msg*)
Send an arbitrary message to a particular client.

Parameters **stream** : `tornado.iostream.IOStream` object

The stream to send the message to.

msg : Message object

The message to send.

Notes

This method can only be called in the IOLoop thread.

Failed sends disconnect the client connection and calls the device on_client_disconnect() method. They do not raise exceptions, but they are logged. Sends also fail if more than self.MAX_WRITE_BUFFER_SIZE bytes are queued for sending, implying that client is falling behind.

send_message_from_thread(*stream*, *msg*)
Thread-safe version of send_message() returning a Future instance.

Returns **A Future that will resolve without raising an exception as soon as** :

the call to send_message() completes. This does not guarantee that the :

message has been delivered yet. If the call to send_message() failed, :

the exception will be logged, and the future will resolve with the :

exception raised. Since a failed call to send_message() will result :

in the connection being closed, no real error handling apart from :

logging will be possible. :

Notes

This method is thread-safe. If called from within the ioloop, send_message is called directly and a resolved tornado.concurrent.Future is returned, otherwise a callback is submitted to the ioloop that will resolve a thread-safe concurrent.futures.Future instance.

setDaemon(*daemonic*)
Set daemonic state of the managed ioloop thread to True / False

Calling this method for a non-managed ioloop has no effect. Must be called before start(), or it will also have no effect

set_ioloop(*ioloop=None*)
Set the tornado IOLoop to use.

Sets the tornado.ioloop.IOLoop instance to use, defaulting to IOLoop.current(). If set_ioloop() is never called the IOLoop is started in a new thread, and will be stopped if self.stop() is called.

Notes

Must be called before start() is called.

start(*timeout=None*)
Install the server on its IOLoop, optionally starting the IOLoop.

Parameters **timeout** : float or None, optional

Time in seconds to wait for server thread to start.

stop(*timeout=1.0*)
Stop a running server (from another thread).

Parameters **timeout** : float or None, optional

Seconds to wait for server to have *started*.

Returns **stopped** : thread-safe Future

Resolves when the server is stopped

wait_running(*timeout=None*)
Wait until the handler thread is running.

class katcp.server.**MessageHandlerThread**(*handler*, *log_inform_formatter*, *logger=<logging.Logger object>*)
Bases: future.types.newobject.newobject

Provides backwards compatibility for server expecting its own thread.

Methods

MessageHandlerThread.isAlive()	
MessageHandlerThread.join([timeout])	Rejoin the handler thread.
MessageHandlerThread.next()	
MessageHandlerThread.on_message(client_conn, msg)	Handle message.
MessageHandlerThread.run()	
MessageHandlerThread.running()	Whether the handler thread is running.
MessageHandlerThread.set_ioloop(ioloop)	
MessageHandlerThread.start([timeout])	
MessageHandlerThread.stop([timeout])	Stop the handler thread (from another thread).
MessageHandlerThread.wait_running([timeout])	Wait until the handler thread is running.

join(*timeout=None*)
Rejoin the handler thread.

Parameters **timeout** : float or None, optional

Time in seconds to wait for the thread to finish.

on_message(*client_conn*, *msg*)
Handle message.

Returns **ready** : Future

A future that will resolve once we're ready, else None.

Notes

on_message should not be called again until *ready* has resolved.

running()
Whether the handler thread is running.

stop(*timeout=1.0*)
Stop the handler thread (from another thread).

Parameters **timeout** : float, optional

Seconds to wait for server to have *started*.

wait_running(*timeout=None*)
Wait until the handler thread is running.

class katcp.server.**ThreadsafeClientConnection**(*server*, *conn_id*)
Bases: *katcp.server.ClientConnection*

Make ClientConnection compatible with messages sent from other threads.

Methods

ThreadsafeClientConnection.disconnect(reason)	Disconnect this client connection for specified reason
ThreadsafeClientConnection.inform(msg)	Send an inform message to a particular client.
ThreadsafeClientConnection.mass_inform(msg)	Send an inform message to all clients.
ThreadsafeClientConnection.next()	
ThreadsafeClientConnection.on_client_disconnect_was_called()	Prevent multiple calls to on_client_disconnect handler.
ThreadsafeClientConnection.reply(reply, orig_req)	Send an asynchronous reply to an earlier request.
ThreadsafeClientConnection.reply_inform(...)	Send an inform as part of the reply to an earlier request.

katcp.server.**construct_name_filter**(*pattern*)
Return a function for filtering sensor names based on a pattern.

Parameters **pattern** : None or str

If None, the returned function matches all names. If pattern starts and ends with '/' the text between the slashes is used as a regular expression to search the names. Otherwise the pattern must match the name of the sensor exactly.

Returns **exact** : bool

Return True if pattern is expected to match exactly. Used to determine whether having no matching sensors constitutes an error.

filter_func : f(str) -> bool

Function for determining whether a name matches the pattern.

katcp.server.**return_future**(*fn*)

Decorator that turns a synchronous function into one returning a future.

This should only be applied to non-blocking functions. Will do set_result() with the return value, or set_exc_info() if an exception is raised.

1.10 Tutorial

1.10.1 Installing the Python Katcp Library

Stable release

To install katcp, run this command in your terminal:

```
$ pip install katcp
```

This is the preferred method to install katcp, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

Alternatively,

```
$ easy_install katcp
```

Note: This requires the setuptools Python package to be installed.

From sources

The sources for katcp can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone https://github.com/ska-sa/katcp-python/
```

Or download the tarball:

```
$ curl -OJL https://github.com/ska-sa/katcp-python/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

1.10.2 Usage

Using the Blocking Client

The blocking client is the most straight-forward way of querying a KATCP device. It is used as follows:

```
from katcp import BlockingClient, Message

device_host = "www.example.com"
device_port = 5000

client = BlockingClient(device_host, device_port)
client.start()
client.wait_protocol() # Optional

reply, informs = client.blocking_request(
    Message.request("help"))

print reply
for msg in informs:
    print msg

client.stop()
client.join()
```

After creating the *BlockingClient* instance, the `start()` method is called to launch the client thread. The `wait_protocol()` method waits until katcp version information has been received from the server, allowing the KATCP version spoken by the server to be known; server protocol information is stores in client.protocol_flags. Once you have finished with the client, `stop()` can be called to request that the thread shutdown. Finally, `join()` is used to wait for the client thread to finish.

While the client is active the *blocking_request()* method can be used to send messages to the KATCP server and wait for replies. If a reply is not received within the allowed time, a `RuntimeError` is raised.

If a reply is received *blocking_request()* returns two values. The first is the *Message* containing the reply. The second is a list of messages containing any KATCP informs associated with the reply.

Using the Callback Client

For situations where one wants to communicate with a server but doesn't want to wait for a reply, the *CallbackClient* is provided:

```
from katcp import CallbackClient, Message

device_host = "www.example.com"
device_port = 5000

def reply_cb(msg):
    print "Reply:", msg

def inform_cb(msg):
    print "Inform:", msg

client = CallbackClient(device_host, device_port)
client.start()
```

(continues on next page)

(continued from previous page)

```
reply, informs = client.callback_request(
    Message.request("help"),
    reply_cb=reply_cb,
    inform_cb=inform_cb,
)

client.stop()
client.join()
```

Note that the `reply_cb()` and `inform_cb()` callback functions are both called inside the client's event-loop thread so should not perform any operations that block. If needed, pass the data out from the callback function to another thread using a `Queue.Queue` or similar structure.

Writing your own Client

If neither the *BlockingClient* nor the *CallbackClient* provide the functionality you need then you can subclass *DeviceClient* which is the base class from which both are derived.

DeviceClient has two methods for sending messages:

- *request()* for sending request *Messages*
- *send_message* for sending arbitrary *Messages*

Internally *request* calls *send_message* to pass messages to the server.

Note: The *send_message()* method does not return an error code or raise an exception if sending the message fails. Since the underlying protocol is entirely asynchronous, the only means to check that a request was successful is receive a reply message. One can check that the client is connected before sending a message using *is_connected()*.

When the *DeviceClient* thread receives a completed message, `handle_message()` is called. The default *handle_message()* implementation calls one of *handle_reply()*, *handle_inform()* or *handle_request()* depending on the type of message received.

Note: Sending requests to clients is discouraged. The `handle_request()` is provided mostly for completeness and to deal with unforseen circumstances.

Each of `handle_reply()`, `handle_inform()` and `handle_request()` dispatches messages to methods based on the message name. For example, a reply message named `foo` will be dispatched to `reply_foo()`. Similarly an inform message named `bar` will be dispatched to `inform_bar()`. If no corresponding method is found then one of `unhandled_reply()`, `unhandled_inform()` or `unhandled_request()` is called.

Your own client may hook into this dispatch tree at any point by implementing or overriding the appropriate methods.

An example of a simple client that only handles replies to `help` messages is presented below:

```
from katcp import DeviceClient, Message
import time

device_host = "www.example.com"
device_port = 5000

class MyClient(DeviceClient):
```

(continues on next page)

(continued from previous page)

```
    def reply_help(self, msg):
        """Print out help replies."""
        print msg.name, msg.arguments

    def inform_help(self, msg):
        """Print out help inform messages."""
        meth, desc = msg.arguments[:2]
        print "---------", meth, "---------"
        print
        print desc
        print "----------------------------"

    def unhandled_reply(self, msg):
        """Print out unhandled replies."""
        print "Unhandled reply", msg.name

    def unhandled_inform(self, msg):
        "Print out unhandled informs."""
        print "Unhandled inform", msg.name


client = MyClient(device_host, device_port)
client.start()

client.request(Message.request("help"))
client.request(Message.request("watchdog"))

time.sleep(0.5)

client.stop()
client.join()
```

Client handler functions can use the *unpack_message()* decorator from *kattypes* module to unpack messages into function arguments in the same way the *request()* decorator is used in the server example below, except that the *req* parameter is omitted.

Using the high-level client API

The high level client API inspects a KATCP device server and presents requests as method calls and sensors as objects.

A high level client for the example server presented in the following section:

```
import tornado

from tornado.ioloop import IOLoop
from katcp import resource_client

ioloop = IOLoop.current()

client = resource_client.KATCPClientResource(dict(
    name='demo-client',
    address=('localhost', 5000),
    controlled=True))

@tornado.gen.coroutine
```

(continues on next page)

(continued from previous page)

```
def demo():
    # Wait until the client has finished inspecting the device
    yield client.until_synced()
    help_response = yield client.req.help()
    print "device help:\n ", help_response
    add_response = yield client.req.add(3, 6)
    print "3 + 6 response:\n", add_response
    # By not yielding we are not waiting for the response
    pick_response_future = client.req.pick_fruit()
    # Instead we wait for the fruit.result sensor status to change to
    # nominal. Before we can wait on a sensor, a strategy must be set:
    client.sensor.fruit_result.set_strategy('event')
    # If the condition does not occur within the timeout (default 5s), we will
    # get a TimeoutException
    yield client.sensor.fruit_result.wait(
        lambda reading: reading.status == 'nominal')
    fruit = yield client.sensor.fruit_result.get_value()
    print 'Fruit picked: ', fruit
    # And see how the ?pick-fruit request responded by yielding on its future
    pick_response = yield pick_response_future
    print 'pick response: \n', pick_response
    # Finally stop the ioloop so that the program exits
    ioloop.stop()

# Note, katcp.resource_client.ThreadSafeKATCPClientResourceWrapper can be used to
# turn the client into a 'blocking' client for use in e.g. ipython. It will turn
# all functions that return tornado futures into blocking calls, and will bounce
# all method calls through the ioloop. In this case the ioloop must be started
# in a separate thread. katcp.ioloop_manager.IOLoopManager can be used to manage
# the ioloop thread.

ioloop.add_callback(client.start)
ioloop.add_callback(demo)
ioloop.start()
```

Writing your own Server

Creating a server requires sub-classing *DeviceServer*. This class already provides all the requests and inform messages required by the KATCP protocol. However, its implementation requires a little assistance from the subclass in order to function.

A very simple server example looks like:

```
import threading
import time
import random

from katcp import DeviceServer, Sensor, ProtocolFlags, AsyncReply
from katcp.kattypes import (Str, Float, Timestamp, Discrete,
                            request, return_reply)

server_host = ""
server_port = 5000

class MyServer(DeviceServer):
```

(continues on next page)

(continued from previous page)

```
    VERSION_INFO = ("example-api", 1, 0)
    BUILD_INFO = ("example-implementation", 0, 1, "")

    # Optionally set the KATCP protocol version and features. Defaults to
    # the latest implemented version of KATCP, with all supported optional
    # features
    PROTOCOL_INFO = ProtocolFlags(5, 0, set([
        ProtocolFlags.MULTI_CLIENT,
        ProtocolFlags.MESSAGE_IDS,
    ]))

    FRUIT = [
        "apple", "banana", "pear", "kiwi",
    ]

    def setup_sensors(self):
        """Setup some server sensors."""
        self._add_result = Sensor.float("add.result",
            "Last ?add result.", "", [-10000, 10000])

        self._time_result = Sensor.timestamp("time.result",
            "Last ?time result.", "")

        self._eval_result = Sensor.string("eval.result",
            "Last ?eval result.", "")

        self._fruit_result = Sensor.discrete("fruit.result",
            "Last ?pick-fruit result.", "", self.FRUIT)

        self.add_sensor(self._add_result)
        self.add_sensor(self._time_result)
        self.add_sensor(self._eval_result)
        self.add_sensor(self._fruit_result)

    @request(Float(), Float())
    @return_reply(Float())
    def request_add(self, req, x, y):
        """Add two numbers"""
        r = x + y
        self._add_result.set_value(r)
        return ("ok", r)

    @request()
    @return_reply(Timestamp())
    def request_time(self, req):
        """Return the current time in seconds since the Unix Epoch."""
        r = time.time()
        self._time_result.set_value(r)
        return ("ok", r)

    @request(Str())
    @return_reply(Str())
    def request_eval(self, req, expression):
        """Evaluate a Python expression."""
        r = str(eval(expression))
        self._eval_result.set_value(r)
```

(continues on next page)

(continued from previous page)

```
        return ("ok", r)

    @request()
    @return_reply(Discrete(FRUIT))
    def request_pick_fruit(self, req):
        """Pick a random fruit."""
        r = random.choice(self.FRUIT + [None])
        if r is None:
            return ("fail", "No fruit.")
        delay = random.randrange(1,5)
        req.inform("Picking will take %d seconds" % delay)

        def pick_handler():
            self._fruit_result.set_value(r)
            req.reply("ok", r)

        self.ioloop.add_callback(
          self.ioloop.call_later, delay, pick_handler)

        raise AsyncReply

    def request_raw_reverse(self, req, msg):
        """
        A raw request handler to demonstrate the calling convention if
        @request decorator are not used. Reverses the message arguments.
        """
        # msg is a katcp.Message.request object
        reversed_args = msg.arguments[::-1]
        # req.make_reply() makes a katcp.Message.reply using the correct request
        # name and message ID
        return req.make_reply('ok', *reversed_args)


if __name__ == "__main__":

    server = MyServer(server_host, server_port)
    server.start()
    server.join()
```

Notice that `MyServer` has three special class attributes `VERSION_INFO`, `BUILD_INFO` and `PROTOCOL_INFO`. `VERSION_INFO` gives the version of the server API. Many implementations might use the same `VERSION_INFO`. `BUILD_INFO` gives the version of the software that provides the device. Each device implementation should have a unique `BUILD_INFO`. `PROTOCOL_INFO` is an instance of `ProtocolFlags` that describes the KATCP dialect spoken by the server. If not specified, it defaults to the latest implemented version of KATCP, with all supported optional features. Using a version different from the default may change server behaviour; furthermore version info may need to be passed to the *`@request`* and *`@return_reply`* decorators.

The `setup_sensors()` method registers *`Sensor`* objects with the device server. The base class uses this information to implement the `?sensor-list`, `?sensor-value` and `?sensor-sampling` requests. *`add_sensor()`* should be called once for each sensor the device should contain. You may create the sensor objects inside `setup_sensors()` (as done in the example) or elsewhere if you wish.

Request handlers are added to the server by creating methods whose names start with "request_". These methods take two arguments – the client-request object (abstracts the client socket and the request context) that the request came from, and the request message. Notice that the message argument is missing from the methods in the example. This is a result of the *`request()`* decorator that has been applied to the methods.

The *request()* decorator takes a list of *KatcpType* objects describing the request arguments. Once the arguments have been checked they are passed in to the underlying request method as additional parameters instead of the request message.

The *return_reply* decorator performs a similar operation for replies. Once the request method returns a tuple (or list) of reply arguments, the decorator checks the values of the arguments and constructs a suitable reply message.

Use of the *request()* and *return_reply()* decorators is encouraged but entirely optional.

Message dispatch is handled in much the same way as described in the client example, with the exception that there are no `unhandled_request()`, `unhandled_reply()` or `unhandled_request()` methods. Instead, the server will log an exception.

Writing your own Async Server

To write a server in the typical tornado async style, modify the example above by adding the following imports

```
import signal
import tornado

from katcp import AsyncDeviceServer
```

Also replace *class MyServer(DeviceServer)* with *class MyServer(AsyncDeviceServer)* and replace the *if __name__ == "__main__":* block with

```
@tornado.gen.coroutine
def on_shutdown(ioloop, server):
    print('Shutting down')
    yield server.stop()
    ioloop.stop()

if __name__ == "__main__":
    ioloop = tornado.ioloop.IOLoop.current()
    server = MyServer(server_host, server_port)
    # Hook up to SIGINT so that ctrl-C results in a clean shutdown
    signal.signal(signal.SIGINT, lambda sig, frame: ioloop.add_callback_from_signal(
        on_shutdown, ioloop, server))
    ioloop.add_callback(server.start)
    ioloop.start()
```

If multiple servers are started in a single ioloop, `on_shutdown()` should be modified to call `stop()` on each server. This is needed to allow a clean shutdown that adheres to the KATCP specification requirement that a *#disconnect* inform is sent when a server shuts down.

Event Loops and Thread Safety

As of version 0.6.0, katcp-python was completely reworked to use Tornado as an event- and network library. A typical Tornado application would only use a single *tornado.ioloop.IOLoop* event-loop instance. Logically independent parts of the application would all share the same ioloop using e.g. coroutines to allow concurrent tasks.

However, to maintain backwards compatibility with the thread-semantics of older versions of this library, it supports starting a *tornado.ioloop.IOLoop* instance in a new thread for each client or server. Instantiating the *BlockingClient* or *CallbackClient* client classes or the *DeviceServer* server class will implement the backward compatible behaviour by default, while using *AsyncClient* or *AsyncDeviceServer* will by default use *tornado.ioloop.IOLoop.current()* as the ioloop (can be overridden using their *set_ioloop* methods), and won't enable thread safety by default (can be overridden using *AsyncDeviceServer.set_concurrency_options()* and *AsyncClient.enable_thread_safety()*)

Note that any message (request, reply, iform) handling methods should not block. A blocking handler will block the ioloop, causing all timed operations (e.g. sensor strategies), network io, etc. to block. This is particularly important when multiple servers/clients share a single ioloop. A good solution for handlers that need to wait on other tasks is to implement them as Tornado coroutines. A *DeviceServer* will not accept another request message from a client connection until the request handler has completed / resolved its future. Multiple outstanding requests can be handled concurrently by raising the *AsyncReply* exception in a request handler. It is then the responsibility of the user to ensure that a reply is eventually sent using the *req* object.

If *DeviceServer.set_concurrency_options()* has *handler_thread=True* (the default for *DeviceServer*, *AsyncDeviceServer* defaults to *False*), all the requests to a server is serialised and handled in a separate request handing thread. This allows request handlers to block without preventing sensor strategy updates, providing backwards-compatible concurrency semantics.

In the case of a purely network-event driven server or client, all user code would execute in the thread context of the server or client event loop. Therefore all handler functions must be non-blocking to prevent unresponsiveness. Unhandled exceptions raised by handlers running in the network event-thread are caught and logged; in the case of servers, an error reply including the traceback is sent over the network interface. Slow operations (such as picking fruit) may be delegated to another thread (if a threadsafe server is used), a callback (as shown in the *request_pick_fruit* handler in the server example) or tornado coroutine.

If a device is linked to processing that occurs independently of network events, one approach would be a model thread running in the background. The KATCP handler code would then defer requests to the model. The model must provide a thread-safe interface to the KATCP code. If using an async server (e.g. *AsyncDeviceServer* or *DeviceServer.set_concurrency_options()* called with *thread_safe=False*), all interaction with the device server needs to be through the `tornado.ioloop.Ioloop.add_callback()` method of the server's ioloop. The server's ioloop instance can be accessed through its *ioloop* attribute. If a threadsafe server (e.g. *DeviceServer* with default concurrency options) or client (e.g. *CallbackClient*) is used, all the public methods provided by this katcp library for sending *!replies* or *#informs* are thread safe.

Updates to *Sensor* objects using the public setter methods are always thread-safe, provided that the same is true for all the observers attached to the sensor. The server observers used to implement sampling strategies are threadsafe, even if an async server is used.

1.10.3 Backwards Compatibility

Server Protocol Backwards Compatibility

A minor modification of the first several lines of the example in *Writing your own Server* suffices to create a KATCP v4 server:

```
from katcp import DeviceServer, Sensor, ProtocolFlags, AsyncReply
from katcp.kattypes import (Str, Float, Timestamp, Discrete,
                            request, return_reply)

from functools import partial
import threading
import time
import random

server_host = ""
server_port = 5000

# Bind the KATCP major version of the request and return_reply decorators
# to version 4
request = partial(request, major=4)
return_reply = partial(return_reply, major=4)
```

(continues on next page)

(continued from previous page)

```
class MyServer(DeviceServer):

    VERSION_INFO = ("example-api", 1, 0)
    BUILD_INFO = ("example-implementation", 0, 1, "")

    # Optionally set the KATCP protocol version as 4.
    PROTOCOL_INFO = ProtocolFlags(4, 0, set([
        ProtocolFlags.MULTI_CLIENT,
    ]))
```

The rest of the example follows as before.

Client Protocol Backwards Compatibility

The *DeviceClient* client automatically detects the version of the server if it can, see *Server KATCP Version Auto-detection*. For a simple client this means that no changes are required to support different KATCP versions. However, the semantics of the messages might be different for different protocol versions. Using the *unpack_message* decorator with *major=4* for reply or inform handlers might help here, although it could use some improvement.

In the case of version auto-detection failing for a given server, *preset_protocol_flags* can be used to set the KATCP version before calling the client's `start()` method.

1.11 How to Contribute

Everyone is welcome to contribute to the katcp-python project. If you don't feel comfortable with writing core katcp we are looking for contributors to documentation or/and tests.

Another option is to report bugs, problems and new ideas as issues. Please be very detailed.

1.11.1 Workflow

A Git workflow with branches for each issue/feature is used.

- There is no special policy regarding commit messages. The first line should be short (50 chars or less) and contain summary of all changes. Additional detail can be included after a blank line.
- Pull requests are normally made to master branch. An exception is when hotfixing a release - in this case the merge target would be to the release branch.

1.11.2 reStructuredText and Sphinx

Documentation is written in reStructuredText and built with Sphinx - it's easy to contribute. It also uses autodoc importing docstrings from the katcp package.

1.11.3 Source code standard

All code should be PEP8 compatible, with more details and exception described in our guidelines.

Note: The accepted policy is that your code **cannot** introduce more issues than it solves!

You can also use other tools for checking PEP8 compliance for your personal use. One good example of such a tool is Flake8 which combines PEP8 and PyFlakes. There are plugins for various IDEs so that you can use your favourite tool easily.

1.11.4 Releasing a new version

From time to time a new version is released. Anyone who wishes to see some features of the master branch released is free to request a new release. One of the maintainers can make the release. The basic steps required are as follows:

Pick a version number

- Semantic version numbering is used: <major>.<minor>.<patch>
- Small changes are done as patch releases. For these the version number should correspond the current development number since each release process finishes with a version bump.
- **Patch release example:**
  - `0.6.3.devN` (current master branch)
  - changes to `0.6.3` (the actual release)
  - changes to `0.6.4.dev0` (bump the patch version at the end of the release process)

Create an issue in Github

- This is to inform the community that a release is planned.
- Use a checklist similar to the one below:

  Task list:
  - [ ] Read steps in the How to Contribute docs for making a release
  - [ ] Edit the changelog and release notes files
  - [ ] Make sure Jenkins tests are still passing on master branch
  - [ ] Make sure the documentation is updated for master (readthedocs)
  - [ ] Create a release tag on GitHub, from master branch
  - [ ] Make sure the documentation is updated for release (readthedocs)
  - [ ] Upload the new version to PyPI
  - [ ] Fill the release description on GitHub
  - [ ] Close this issue

- A check list is this form on github can be ticked off as the work progresses.

Make a branch from `master` to prepare the release

- Example branch name: `user/ajoubert/prepare-v0.6.3`.
- Edit the `CHANGELOG` and release notes (in `docs/releasenotes.rst`). Include *all* pull requests since the previous release.
- Create a pull request to get these changes reviewed before proceeding.

Make sure Jenkins is OK on `master` branch

- All tests on Jenkins must be passing. If not, bad luck - you'll have to fix it first, and go back a few steps...

Make sure the documentation is ok on master

- Log in to https://readthedocs.org.
- Get account permissions for https://readthedocs.org/projects/katcp-python from another maintainer, if necessary.
- **Readthedocs *should* automatically build the docs for each:**
  - push to master (latest docs)
  - new tags (e.g v0.6.3)
- **If it doesn't work automatically, then:**
  - Trigger manually here: https://readthedocs.org/projects/katcp-python/builds/

Create a release tag on GitHub

- On the Releases page, use "Draft a new release".
- Tag must match the format of previous tags, e.g. `v0.6.3`.
- Target must be the `master` branch.

Make sure the documentation is updated for the newly tagged release

- **If the automated build doesn't work automatically, then:**
  - Trigger manually here: https://readthedocs.org/projects/katcp-python/builds/
- Set the new version to "active" here: https://readthedocs.org/dashboard/katcp-python/versions/

Upload the new version to PyPI

- Log in to https://pypi.org.
- Get account permissions for katcp from another contributor, if necessary.
- If necessary, pip install twine: https://pypi.org/project/twine/
- **Build update from the tagged commit:**
  - `$ git clean -xfd # Warning - remove all non-versioned files and directories`
  - `$ git fetch`
  - `$ git checkout v0.6.3`
  - `$ python setup.py sdist bdist_wheel`
  - `$ python3 setup.py bdist_wheel`
- **Upload to testpypi, and make sure all is well:**
  - `$ twine upload -r testpypi dist/katcp-0.6.3.tar.gz`
  - `$ twine upload -r testpypi dist/katcp-0.6.3-py2-none-any.whl`
  - `$ twine upload -r testpypi dist/katcp-0.6.3-py3-none-any.whl`
- **Test installation (in a virtualenv):**
  - `$ pip install katcp`
  - `$ pip install -U -i https://test.pypi.org/simple/ katcp`
- **Upload the source tarball and wheel to the real PyPI:**

  - `$ twine upload dist/katcp-0.6.3.tar.gz`
  - `$ twine upload dist/katcp-0.6.3-py2-none-any.whl`
  - `$ twine upload dist/katcp-0.6.3-py3-none-any.whl`

Fill in the release description on GitHub

- Content must be the same as the details in the changelog.

Close off release issue in Github

- All the items on the check list should be ticked off by now.
- Close the issue.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

k

Index

D

I

J

K

L

M

N

O

P

R

S

T

U

V

W