streamkinect2 Documentation

Release

Rich Wareham

September 18, 2014

Contents

Contents:

CHAPTER 1

Example programs

Here are some example programs which use the `streamkinect2` API.

1.1 Simple ping client

The following program shows how to use the `streamkinect2.server.ServerBrowser` class to discover servers on the network. For each server, a simple client is created which sends a `ping` to the server and logs when a `pong` is received.

```
#!/usr/bin/env python
"""
Simple client which pings each server as it is discovered.

"""
import logging
import threading

from tornado.ioloop import IOLoop
from streamkinect2.server import ServerBrowser
from streamkinect2.client import Client

# Install the zmq ioloop
from zmq.eventloop import ioloop
ioloop.install()

# Get our logger
log = logging.getLogger(__name__)

# Our listening class
class Listener(object):
    def __init__(self, browser, io_loop = None):
        self.clients = {}
        self.io_loop = io_loop or IOLoop.instance()
        browser.on_add_server.connect(self.add_server, sender=browser)
        browser.on_remove_server.connect(self.remove_server, sender=browser)

        # Keep a reference to browser since we remain interested and do not
        # wish it garbage collected.
        self.browser = browser

    def add_server(self, browser, server_info):
        log.info('Discovered server "{0.name}" at "{0.endpoint}"'.format(server_info))
```

```
        client = Client(server_info.endpoint, connect_immediately=True)
        self.clients[server_info.endpoint] = client

        def pong(server_info=server_info):
            log.info('Got pong from "{0.name}"'.format(server_info))
            self.clients[server_info.endpoint].disconnect()
            del self.clients[server_info.endpoint]

        log.info('Pinging server "{0.name}"...'.format(server_info))
        client.ping(pong)

    def remove_server(self, browser, server_info):
        log.info('Server "{0.name}" at "{0.endpoint}" went away'.format(server_info))

class IOLoopThread(threading.Thread):
    def run(self):
        # Create the server browser
        log.info('Creating server browser...')
        listener = Listener(ServerBrowser())

        # Run the ioloop
        log.info('Running...')
        ioloop.IOLoop.instance().start()

        log.info('Stopping')

    def stop(self):
        io_loop = ioloop.IOLoop.instance()
        io_loop.add_callback(io_loop.stop)
        self.join(3)

def main():
    # Set log level
    logging.basicConfig(level=logging.INFO)

    print('==========================================')
    print('Press Enter to exit')
    print('==========================================')

    # Start the event loop
    ioloop_thread = IOLoopThread()
    ioloop_thread.start()

    # Wait for input
    input()

    # Stop thread
    ioloop_thread.stop()

if __name__ == '__main__':
    main()
```

1.2 Mock kinect server

The following program shows how to create a simple server which will serve data from a mock Kinect. See the `streamkinect2.mock` module.

```
#!/usr/bin/env python
"""
Simple server using the mock Kinect.

"""
import logging
import threading

from streamkinect2.server import Server
from streamkinect2.mock import MockKinect

# Install the zmq ioloop
from zmq.eventloop import ioloop
ioloop.install()

# Get our logger
log = logging.getLogger(__name__)

class IOLoopThread(threading.Thread):
    def run(self):
        # Create the server
        log.info('Creating server')
        server = Server()

        # Add mock kinect device to server
        kinect = MockKinect()
        server.add_kinect(kinect)

        # With the server and kinect running...
        log.info('Running server...')
        with server, kinect:
            # Run the ioloop
            ioloop.IOLoop.instance().start()

        # The server has now stopped
        log.info('Stopped')

    def stop(self):
        io_loop = ioloop.IOLoop.instance()
        io_loop.add_callback(io_loop.stop)
        self.join(3)

def main():
    # Set log level
    logging.basicConfig(level=logging.INFO)

    print('================================================')
    print('Press Enter to exit')
    print('================================================')

    # Start the event loop
    ioloop_thread = IOLoopThread()
    ioloop_thread.start()
```

```
    # Wait for input
    input()

    # Stop thread
    ioloop_thread.stop()

if __name__ == '__main__':
    main()
```

CHAPTER 2

Network protocol

The network protocol is based on Zeroconf for server discovery and ZeroMQ for communication. The architecture is a traditional client-server model with one server dealing with zero, one or many clients. In practice there will probably be one server and one client. The transport is based entirely on ZeroMQ sockets and so it is recommended that one read some of the ZeroMQ guide before this document.

2.1 Server discovery

Servers advertise themselves over ZeroConf using the `_kinect2._tcp` service type. The IP address and port associated with that service is converted into a ZeroMQ endpoint as `tcp://<address>:<port>` and is used to find the "control" endpoint of the server.

2.2 Endpoints

Much like the USB protocol, each server advertises a number of "endpoints" which are specified as a ZeroMQ address, usually of the form `tcp://<address>:<port>`. The "control" endpoint is advertised over ZeroConf and may be used to query other endpoints. An endpoint is usually a ZeroMQ socket pair, one on the client and one on the server.

2.2.1 Control Endpoint

The "control" endpoint is a REP socket on the server which expects to be connected to via a REQ socket on the client. Clients initiate communication by sending a `who` message. The server will then respond with a `me` message. The client may then send other messages expecting each time a reply from the server. This is repeated until the client disconnects.

All messages are multipart messages with one or two frames. The first frame is a single byte which indicates the message type. The second frame, if present, represents a JSON encoded object which is the "payload" of the message.

Each message type has its own semantics and payload schema. Some messages may only be sent by a client and some only by a server.

`error` type

An `error` message (type 0x00) MUST only be sent by the server. The server MAY send an `error` message in reply to any incoming request. The payload must contain a `reason` field with a human-readable description of the error. The client MAY choose to disconnect from the server or silently ignore the error.

`ping` type

A `ping` message (type 0x01) MUST only be sent by a client. No payload is required. The server MUST respond with an empty-payload message of type `pong` or an `error` message.

`pong` type

A `pong` message (type 0x02) MUST only be sent by a server. It MUST do so in response to a `ping` if no `error` is sent. No payload is required.

`who` type

A `who` message (type 0x03) MUST only be sent by a client. No payload is required. The server MUST respond with a `me` message or an `error` message.

`me` type

A `me` messages MUST only be sent by a server. It MUST do so in response to a `who` message if no `error` is sent. A payload MUST be present. The payload MUST be an object including at least a `version` field which should be the numeric value 1. A client MUST ignore any `me` message with a `version` field set to any other value.

The payload MUST include a field named `name` whose value is a string representing a human-readable name for the server.

The payload MUST include a field named `endpoints` whose value is an object whose fields correspond to endpoint names and whose values correspond to ZeroMQ-style endpoint addresses. The client MUST ignore any endpoints whose name it does not recognise. The server MAY advertise any endpoints it wishes but it MUST include at least a `control` endpoint with a ZeroMQ address corresponding to the control endpoint. The advertised endpoints MAY be non-unique and MAY have different IP addresses.

The payload MUST include a field named `devices` whose value is an array of device records. A device record is a JSON object. A device record MUST include a field named `id` whose value is a string giving a unique name for a Kinect connected to the server. A device record MUST include a field named `endpoints` whose value takes the same format (but not necessarily the same value) as the `endpoints` object in the payload. This `endpoints` object gives endpoints which are specific to a particular device.

A typical payload will look like the following:

```
{
    "version": 1,
    "name": "Bob's Kinect",
    "endpoints": {
        "control": "tcp://10.0.0.1:1234"
    },
    "devices": [
        {
            "id": "123456789abcdefghijklmnopqrstuv",
            "endpoints": {
                "depth": "tcp://10.0.0.1:1236"
            }
        }
    ],
}
```

CHAPTER 3

API Reference

Kinect2 streaming server and client

3.1 Event handling

Some `streamkinect2` object emit events. The blinker library is used to handle signals. See the blinker documentation for full details. As an example, here is how to register an event handler for a new depth frame from a `streamkinect2.mock.MockKinect` object:

```
from streamkinect2.mock import MockKinect

kinect = MockKinect()

# The "depth_frame" argument name is important here as the depth frame is
# passed as a keyword argument.
def handler_func(kinect, depth_frame):
    print('New depth frame')

MockKinect.on_depth_frame.connect(handler_func, kinect)
```

Alternatively, one may use the `connect_via()` decorator:

```
from streamkinect2.mock import MockKinect

kinect = MockKinect()

@MockKinect.on_depth_frame.connect_via(kinect)
def handler_func(kinect, depth_frame):
    print('New depth frame')
```

Note that, by default, signal handlers are kept as weak references so that they do not need to be explicitly disconnected before they can be garbage collected.

3.2 Common elements for both client and server

class `streamkinect2.common.`**`EndpointType`**
Enumeration of endpoints exposed by a `Server`.

`control`
A *REP* endpoint which accepts JSON-formatted control messages.

depth
A *PUB* endpoint which broadcasts compressed depth frames to connected subscribers.

exception streamkinect2.common.**ProtocolError**
Raised when some low-level error in the network protocol has been detected.

3.3 Server

class streamkinect2.server.**Server**(*address=None*, *start_immediately=False*, *name=None*, *zmq_ctx=None*, *io_loop=None*, *announce=True*)
A server capable of streaming Kinect2 data to interested clients.

Servers may have their lifetime managed by using them within a `with` statement:

```
with Server() as s:
    # server is running
    pass
# server has stopped
```

address and *port* are the bind address (as a decimal-dotted IP address) and port from which to start serving. If *port* is None, a random port is chosen. If *address* is *None* then attempt to infer a sensible default.

name should be some human-readable string describing the server. If *None* then a sensible default name is used.

zmq_ctx should be the zmq context to create servers in. If *None*, then `zmq.Context.instance()` is used to get the global instance.

If not *None*, *io_loop* is the event loop to pass to `zmq.eventloop.zmqstream.ZMQStream` used to communicate with the cleint. If *None* then global IOLoop instance is used.

If *announce* is True then the server will be announced over ZeroConf when it starts running.

address
The address bound to as a decimal-dotted string.

endpoints
The zeromq endpoints for this server. A *dict*-like object keyed by endpoint type. (See `streamkinect2.common.EndpointType`.)

is_running
True when the server is running, *False* otherwise.

kinects
`list` of kinect devices managed by this server. See `add_kinect()`.

add_kinect(*kinect*)
Add a Kinect device to this server. *kinect* should be a object implementing the same interface as `streamkinect2.mock.MockKinect`.

remove_kinect(*kinect*)
Remove a Kinect device previously added via `add_kinect()`.

start()
Explicitly start the server. If the server is already running, this has no effect beyond logging a warning.

stop()
Explicitly stop the server. If the server is not running this has no effect beyond logging a warning.

class streamkinect2.server.**ServerBrowser**(*io_loop=None*, *address=None*)
An object which listens for kinect2 streaming servers on the network. The object will keep listening as long as it is alive and so if you want to continue to receive notification of servers, you should keep it around.

io_loop is an instance of `tornado.ioloop.IOLoop` which should be used to schedule sending signals. If *None* then the global instance is used. This is needed because server discovery happens on a separate thread to the tornado event loop which is used for the rest of the network communication. Hence, when a server is discovered, the browser co-ordinates with the event loop to call the `add_server()` and `remove_server()` methods on the main IOLoop thread.

address is an explicit bind IP address for an interface to listen on as a decimal-dotted string or *None* to use the default.

on_add_server = <blinker.base.Signal object at 0x7f4cbf2241d0>
Signal emitted when a new server is discovered on the network. Receivers should take a single keyword argument, *server_info*, which will be an instance of `ServerInfo` describing the server.

on_remove_server = <blinker.base.Signal object at 0x7f4cbf224290>
Signal emitted when a server removes itself from the network. Receivers should take a single keyword argument, *server_info*, which will be an instance of `ServerInfo` describing the server.

class `streamkinect2.server.`**ServerInfo**
Kinect2 Stream server information.

This is a subclass of the bultin `tuple` class with named accessors for convenience. The tuple holds *name*, *endpoint* pairs.

name
A server-provided human-readable name for the server.

endpoint
Connection information for control channel which should be passed to `streamkinect2.client.Client`.

3.4 Client

class `streamkinect2.client.`**Client**(*control_endpoint*, *connect_immediately=False*, *zmq_ctx=None*, *io_loop=None*)
Client for a streaming kinect2 server.

Usually the client will be used with a `with` statement:

```
with Client(endpoint) as c:
    # c is connected here
    pass
# c is disconnected here
```

control_endpoint is the zeromq control endpoint for the server which should be connected to.

If not *None*, *zmq_ctx* is the zeromq context to create sockets in. If *zmq_ctx* is *None*, the global context returned by `zmq.Context.instance()` is used.

If not *None*, *io_loop* is the event loop to pass to `zmq.eventloop.zmqstream.ZMQStream` used to listen to responses from the server. If *None* then global IO loop is used.

If *connect_immediately* is *True* then the client attempts to connect when constructed. If *False* then `connect()` must be used explicitly.

server_name
A string giving a human-readable name for the server or *None* if the server has not yet replied to our initial query.

endpoints
A `dict` of endpoint addresses keyed by `streamkinect2common.EndpointType`.

is_connected
True if the client is connected. *False* otherwise.

The following attributes are mostly of use to the unit tests and advanced users.

heartbeat_period
The delay, in milliseconds, between "heartbeat" requests to the server. These are used to ensure the server is still alive. Changes to this attribute are ignored once `connect()` has been called.

response_timeout
The maximum wait time, in milliseconds, the client waits for the server to reply before giving up.

connect()
Explicitly connect the client.

disconnect()
Explicitly disconnect the client.

enable_depth_frames(*kinect_id*)
Enable streaming of depth frames. *kinect_id* is the id of the device which should have streaming enabled.

Raises ValueError if *kinect_id* does not correspond to a connected device

on_add_kinect = <blinker.base.Signal object at 0x7f4cbf0c9590>
A signal which is emitted when a new kinect device is available. Handlers should accept a single keyword argument *kinect_id* which is the unique id associated with the new device.

on_connect = <blinker.base.Signal object at 0x7f4cbf0c9510>
A signal which is emitted when the client connects to a server.

on_depth_frame = <blinker.base.Signal object at 0x7f4cbf0c9610>
A signal which is emitted when a new depth frame is available. Handlers should accept two keyword arguments: *depth_frame* which will be an instance of an object with the same interface as `DepthFrame` and *kinect_id* which will be the unique id of the kinect device producing the depth frame.

on_disconnect = <blinker.base.Signal object at 0x7f4cbf0c9550>
A signal which is emitted when the client disconnects from a server.

on_remove_kinect = <blinker.base.Signal object at 0x7f4cbf0c95d0>
A signal which is emitted when a kinect device is removed. Handlers should accept a single keyword argument *kinect_id* which is the unique id associated with the new device.

ping(*pong_cb=None*)
Send a 'ping' request to the server. If *pong_cb* is not *None*, it is a callable which is called with no arguments when the pong response has been received.

3.5 Depth frame compression

class `streamkinect2.compress.`**DepthFrameCompressor**(*kinect*, *io_loop=None*)
Asynchronous compression pipeline for depth frames.

kinect is a `streamkinect2.mock.MockKinect`-like object. Depth frames emitted by `on_depth_frame()` will be compressed with frame-drop if the compressor becomes overloaded.

If *io_loop* is provided, it specifies the `tornado.ioloop.IOLoop` which is used to co-ordinate the worker process. If not provided, the global instance is used.

kinect
Kinect object associated with this compressor.

on_compressed_frame = <blinker.base.Signal object at 0x7f4cbf217d90>
Signal emitted when a new compressed frame is available. Receivers take a single keyword argument, *compressed_frame*, which is a Python buffer-like object containing the compressed frame data. The signal is emitted on the IOLoop thread.

3.6 Mock kinect

Note: This module requires `numpy` to be installed.

Support for a mock kinect when testing.

class `streamkinect2.mock.`**DepthFrame**
A single frame of depth data.

data
Python buffer-like object pointing to raw frame data as a C-ordered array of uint16.

shape
Pair giving the width and height of the depth frame.

class `streamkinect2.mock.`**MockKinect**
A mock Kinect device.

This class implements a "virtual" Kinect which generates some mock data. It can be used for testing or benchmarking.

Use `start()` and `stop()` to start and stop the device or wrap it in a `with` statement:

```
with MockKinect() as kinect:
    # kinect is running here
    pass
# kinect has stopped running
```

Note: Listener callbacks are called in a separate thread. If using something like `tornado.ioloop.IOLoop`, then you will need to make sure that server messages are sent on the right thread. The `streamkinect2.server.Server` class should take care of that in most cases you will encounter.

unique_kinect_id
A string with an opaque, unique id for this Kinect.

on_depth_frame = <blinker.base.Signal object at 0x7f4cbf061050>
A signal which is emitted when a new depth frame is available. Handlers should accept a single keyword argument *depth_frame* which will be an instance of `DepthFrame`.

start()
Start the mock device running. Mock data is generated on a separate thread.

stop()
Stop the mock device running. Blocks until the thread shuts down gracefully with a one second timeout.

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*

S