Arkady

Apr 13, 2019

Contents:

CHAPTER 1

Dependencies

Arkady uses Python3's built-in *asyncio*, so it supports and requires use of Python3.5+.

ZeroMQ is employed for socket communication, so pyzmq is required.

pyzmq

CHAPTER 2

What Arkady IS

The central problem Arkady seeks to solve is how to set up an interface to an arbitrary "device" and control it from another process. This can be local or remote over a network; it uses ZeroMQ socket communication which is robust and lightweight.

CHAPTER 3

What Arkady IS NOT

Though the Arkady library may provide some utilities for talking to Arkady applications. It does not intend to be the central means by which you control said applications. Not because Arkady is lazy, but because Arkady wants to give you freedom. Because ZeroMQ sockets are used for communication, you can communicate with Arkady application interfaces in most major languages: Java, C++, Python, Javascript… all good!

CHAPTER 4

What can I use Arkady to do?

You can use Arkady to separate the controller logic of a piece of software from the nitty-gritty of hardware integration. This problem is why I wrote the code that turned into Arkady in the first place: I had an application that needed to simultaneously interact with Arduinos, DMX, video, audio, sensors, and keep track of program control flow. Using Arkady I was able to create a simple interface to all my devices in one program, and to write clean logic in another program to leverage this interface.

You can use Arkady to put a network interface on a hardware device and save a lot of wiring. Today you can get a Raspberry Pi Zero W for 5 USD, with a bit more added for peripherals, you can put almost anything with wired control onto the network with Arkady economically.

CHAPTER 5

Creating an Arkady interface

Suppose I wish to be able to read the temperature of my Raspberry Pi from another computer on my network. This command would do the trick from the command line: `/opt/vc/bin/vcgencmd measure_temp` so I want to set up an Arkady *device* for it.

```
from arkady.devices import AsyncDevice
import subprocess

class RpiCPUTemp(AsyncDevice):
    def handler(self, msg, *args, **kwargs):
        if msg == 'get':
            # command returns bytestring like b"temp=47.8'C"
            temp_out =  subprocess.run(
                ['/opt/vc/bin/vcgencmd',
                 'measure_temp'],
                capture_output=True).stdout.decode('utf-8')
            # extract temperature string
            temperature = temp_out.split('=')[1].rstrip()
            return temperature
        else:
            return 'Unrecognized msg. Must be "get"'
```

Now I need to create an Arkady application to make use of this custom "device".

```
from arkady import Application

class RpiCPUTempApp(Application):
    def config(self):
        """This is called as the last step in setup for the Application"""
        # Creates the device and gives it the name 'temp'
        self.add_device(RpiCPUTemp, 'temp')
        # Creates a router type listener and listens on port 5555
        self.add_router(bind_to='tcp://*:5555')
```

(continues on next page)

(continued from previous page)

```
my_app = RpiCPUTempApp()
my_app.run()  # blocks until terminated
```

So now this application will wait for messages. Any message beginning with the word *temp* will be referred to the *RPiCPUTemp* device. The message after the name *temp* will be give to the device method *handler* as the *msg* argument. *RPiCPUTemp.handler* only recognizes the message *"get"* and will report an error if it gets something else. Otherwise it runs the command and returns the temperature string.

Now, you can send messages via ZeroMQ in whatever language you please. Here's a simple program in Python that will do so.

```
import time
import zmq

RPI_URI = 'tcp://localhost:5555'  # Same machine
# RPI_URI = 'tcp://192.168.1.111:5555'  # remote machine

context = zmq.Context()
socket = context.socket(zmq.REQ)  # Request type socket, expects replies
socket.connect(RPI_URI)

while True:
    # Send 'temp get'. First word is device name, remainder is message
    socket.send_string('temp get')
    # Requests (must) receive replies. Print our reply
    print(socket.recv_string())
    time.sleep(5)  # Sleep 5 seconds between temperature checks
```

CHAPTER 6

Sphinx documentation contents

6.1 Arkady Devices

"Device" is a loose term in Arkady. It represents a fundamental unit of interface and should generally map to a logical unit of control. This could be interaction with an actual physical device or peripheral such as a sensor, a motor, an Arduino, a DMX controller, etc. Or it could be something more virtual such as a set of system calls, internet/intranet queries, a managed subprocess and more.

Two basic device patterns are implemented: *SerialDevice* and *AsyncDevice*. Use of *SerialDevice* is recommended when the underlying work must be strictly serial (meaning non-parallel). *AsyncDevice* is suitable when multiple executions of the *handler* can safely run simultaneously.

class arkady.devices.**AsyncDevice**(**args*, ***kwargs*)

requests_runner()
Responsible for taking jobs out of the jobs queue and executing them.

Not implemented in this base class, must be overridden.

Returns

class arkady.devices.**Device**(**args*, *loop=None*, ***kwargs*)
The Base Device from which all other devices derive, whether they have synchronous or asynchronous underlying work.

requests_runner()
Responsible for taking jobs out of the jobs queue and executing them.

Not implemented in this base class, must be overridden.

Returns

class arkady.devices.**DummyAsyncDevice**(**args*, ***kwargs*)

class arkady.devices.**DummySerialDevice**(**args*, ***kwargs*)

class arkady.devices.**SerialDevice**(**args*, ***kwargs*)

requests_runner()
Responsible for taking jobs out of the jobs queue and executing them.

Not implemented in this base class, must be overridden.

Returns

6.2 Arkady Listeners

arkady.listeners.**router**(*application*, *bind_to=None*)
The `router` listener handles asynchronous requests in the request-reply pattern. A request of type *zmq.REQ* shall be given a reply of type *zmq.REP*

Parameters

- **application** –
- **bind_to** (*string*) – Network path on which to listen. Defaults to `'tcp://*:5555'`

Returns

arkady.listeners.**sub**(*application*, *connect_to=None*, *topics=None*)
The `sub` listener handles asynchronous requests in the pub-sub pattern. A request of type *zmq.PUB* receives no reply

Parameters

- **application** –
- **connect_to** (*string*) – A well-known network URI, like 'tcp://192.168.1.200:5555'
- **topics** (*[string]*) – A list of topics as to subscribe to

Returns

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index