Arkady

Apr 16, 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.

CHAPTER 2

What Arkady IS

The central problem Arkady seeks to solve is how to set up an interface to an arbitrary "component" 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 components 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 component 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 *component* for it.

```
from arkady.components import AsyncComponent
import subprocess

class RpiCPUTemp(AsyncComponent):
    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 "component".

```
from arkady import Application

class RpiCPUTempApp(Application):
    def config(self):
        """This is called as the last step in setup for the Application"""
        # Creates the component and gives it the name 'temp'
        self.add_component(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* component. The message after the name *temp* will be give to the component 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 component 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 Installing Arkady

Arkady is still in early development and I have not issued a release on PyPI, so downloading and installing from source on GitHub is the recommended course of action currently.

https://github.com/SavinaRoja/arkady

Arkady should be installed with Python of version 3.5 or greater, as this is when `asyncio` was introduced. Arkady makes use of `asyncio` internally for concurrent, non-blocking function. You can write Arkady components to take advantage of `asyncio`, but more on that later. All references to `python` and `pip` in commands are assumed to be for that version or later.

Installation is simple once you have downloaded the source, just navigate to the base directory of the source code and perform:

```
python setup.py install
```

or

```
pip install .
```

6.1.1 Development Installation

If you are looking to experiment with modifying Arkady, you will want to install it in so-called "development mode" using the `-e` flag Creating and working in a virtual environment is simple and recommended, here it is in bash:

```
python -m venv venv
source venv/bin/activate
pip install -e .
```

and for Windows:

```
python -m venv venv
.\venv\Scripts\activate
pip install -e .
```

6.2 Getting Started with Arkady

The first step in working with Arkady is to think about what "action" you wish to make available for networked (websocket from another machine) or simply interprocess control. Let's start with something simple and accessible, a generic interface to Nanpy, all you need to work along is an Arduino.

Nanpy is a great prototyping tool that lets you interface with an Arduino over a serial connection to control and read pins. Setup and usage will be covered in more detail below.

6.2.1 Choosing a listener

A listener is responsible for "listening" for external input, and at present there are two in Arkady: `router` and `sub`. The *router* handles asynchronous request-reply interaction and is best to use when you need to return information to a requester, or at least provide acknowledgment of receipt. The *sub* handles publish-subscribe type interaction which is a one-way kind of communication.

Because Nanpy permits reading the value of a pin, and I'll want to be able to send this data back to a requester, I'm going to add a router to my Arkady Application.

```
# my_application.py

from arkady import Application


class MyApplication(Application):
    def config(self):
        self.add_router(bind_to='tcp://*:5555')
```

The method `config` gets called during creation of the application and is a good place to put registrations of listeners and (as we'll address in a moment) components. `add_router` will set up a router listener for the application, and we have explicitly passed `bind_to='tcp://*:5555'` which instructs the added router to listen on port 5555 (this is also the default if you don't specify).

This application alone won't do anything until we add at least one component to it.

6.2.2 Creating a component

The Nanpy interface has four main functions which we will wish to make accessible through the application: `digitalRead`, `analogRead`, `digitalWrite`, and `analogWrite`. So let's create a component that implements those actions.

There are two central base components in Arkady from which to derive: `SerialComponent` and `AsyncComponent`. Arkady handles concurrency so it's possible for more than one message to come in for a component and run simultaneously. This can be a problem in some cases, as in this case where we should only allow one message exchange over the USB to the Arduino at one time. So we'll choose `SerialComponent` which ensures serial (not concurrent) execution of jobs.

```
from arkady.components import SerialComponent
from nanpy import SerialManager, ArduinoApi


class GenericNanpy(SerialComponent):
    def __init__(self, port, *args, **kwargs):
        super(GenericNanpy, self).__init__(*args, **kwargs)
        self._serial_manager = SerialManager(device=port, baudrate=115200)
        self.ardu = ArduinoApi(self._serial_manager)

    def analog_read(self, pin_number, *_words):
        """Read the pin in analog mode"""
        self.ardu.pinMode(pin_number, self.ardu.INPUT)
        return self.ardu.analogRead(pin_number)

    def digital_read(self, pin_number, *_words):
        """Read the pin in digital mode"""
        self.ardu.pinMode(pin_number, self.ardu.INPUT)
        return self.ardu.digitalRead(pin_number)

    def analog_write(self, pin_number, value, *_words):
        """Write the pin in analog mode to value"""
        try:
            value = int(value)
        except ValueError:
            return 'ERROR: Got a value that could not be treated as integer'
        self.ardu.pinMode(pin_number, self.ardu.OUTPUT)
        self.ardu.analogWrite(pin_number, value)

    def digital_write(self, pin_number, value, *_words):
        """Write the pin HIGH if value is 'high' otherwise LOW."""
        self.ardu.pinMode(pin_number, self.ardu.OUTPUT)
        if value == 'high':
            self.ardu.digitalWrite(pin_number, self.ardu.HIGH)
        else:
            self.ardu.digitalWrite(pin_number, self.ardu.LOW)
```

In the `__init__` method we initialize the Nanpy `ArduinoApi` on the specified port In the methods `analog_read`, `digital_read`, `analog_write`, and `digital_write` we provide Nanpy functionality.

The next step is to write a message handler for the component. The application's listeners will pass messages received along to this method.

```
class GenericNanpy(SerialComponent):

    ...

    def handler(self, msg, *args, **kwargs):
        """Handle an inbound message. Returned values go back as reply."""
        word_map = {
            'dwrite': self.digital_write,
            'awrite': self.analog_write,
            'dread': self.digital_read,
            'aread': self.analog_read,
        }
        words = msg.split()
        if len(words) < 2:  # Check for too short message
```

(continues on next page)

(continued from previous page)

```
            return 'ERROR: message must contain at least 2 words!'
        key_word = words[0]
        try:
            pin = int(words[1])
        except ValueError:
            return 'ERROR: got non-int for pin number {}'.format(words[1])
        if key_word not in word_map:  # Check if we recognize the first word
            return 'ERROR: not one of the known functions, {}'.format(word_map.keys())
        try:
            # Call the corresponding method
            ret_val = word_map[key_word](pin, *words[2:])
            if ret_val is not None:
                ret_val = str(ret_val)
            return ret_val
        except:
            return 'ERROR: "{}" failed, maybe a bad message or connection'.format(msg)
```

This `handler` method does the job of interpreting messages so that action may be taken, along with some error handling. Care is taken to return the results of the called methods, as the returned string values will get passed back to a client of our Arkady application as the body of a reply message, this will be addressed further below.

Now that our custom component has been implemented, we wish to add it to our application and register it so that messages may be passed to it. Let's update `my_application.py`:

```
# my_application.py

from arkady import Application

ARDUINO_PORT = '/dev/ttyUSB0'  # On Windows this is more like "COM3"


class MyApplication(Application):
    def config(self):
        self.add_component('nanpy', GenericNanpy, ARDUINO_PORT)
        self.add_router(bind_to='tcp://*:5555')
```

This addition to `config` tells the application that when the listeners receive messages, if the first word of the message is "nanpy" then the message should go to an instance of `GenericNanpy`, created once for the application with `GenericNanpy(ARDUINO_PORT)`.

Now the application is completed, and here it is all together:

```
#!/usr/bin/env python3

"""
Demonstration of a very generic Arkady interface to NanPy.

Direct dependencies are: arkady, nanpy
Indirect depencences are: pyserial, pyzmq
"""

from arkady import Application
from arkady.components import SerialComponent

from nanpy import SerialManager, ArduinoApi

ARDUINO_PORT = '/dev/ttyUSB0'  # On Windows this is more like "COM3"
```

(continues on next page)

(continued from previous page)

```
class GenericNanpy(SerialComponent):
    def __init__(self, port, *args, **kwargs):
        super(GenericNanpy, self).__init__(*args, **kwargs)
        self._serial_manager = SerialManager(device=port, baudrate=115200)
        self.ardu = ArduinoApi(self._serial_manager)

    def analog_read(self, pin_number, *_words):
        """Read the pin in analog mode"""
        self.ardu.pinMode(pin_number, self.ardu.INPUT)
        return self.ardu.analogRead(pin_number)

    def digital_read(self, pin_number, *_words):
        """Read the pin in digital mode"""
        self.ardu.pinMode(pin_number, self.ardu.INPUT)
        return self.ardu.digitalRead(pin_number)

    def analog_write(self, pin_number, value, *_words):
        """Write the pin in analog mode to value"""
        try:
            value = int(value)
        except ValueError:
            return 'ERROR: Got a value that could not be treated as integer'
        self.ardu.pinMode(pin_number, self.ardu.OUTPUT)
        self.ardu.analogWrite(pin_number, value)

    def digital_write(self, pin_number, value, *_words):
        """Write the pin HIGH if value is 'high' otherwise LOW."""
        self.ardu.pinMode(pin_number, self.ardu.OUTPUT)
        if value == 'high':
            self.ardu.digitalWrite(pin_number, self.ardu.HIGH)
        else:
            self.ardu.digitalWrite(pin_number, self.ardu.LOW)

    def handler(self, msg, *args, **kwargs):
        """Handle an inbound message. Returned values go back as reply."""
        word_map = {
            'dwrite': self.digital_write,
            'awrite': self.analog_write,
            'dread': self.digital_read,
            'aread': self.analog_read,
        }
        words = msg.split()
        if len(words) < 2:  # Check for too short message
            return 'ERROR: message must contain at least 2 words!'
        key_word = words[0]
        try:
            pin = int(words[1])
        except ValueError:
            return 'ERROR: got non-int for pin number {}'.format(words[1])
        if key_word not in word_map:  # Check if we recognize the first word
            return 'ERROR: not one of the known functions, {}'.format(word_map.keys())
        try:
            # Call the corresponding method
            ret_val = word_map[key_word](pin, *words[2:])
            if ret_val is not None:
```

(continues on next page)

(continued from previous page)

```
                ret_val = str(ret_val)
            return ret_val
        except:
            return 'ERROR: "{}" failed, maybe a bad message or connection'.format(msg)


class MyApplication(Application):
    def config(self):
        self.add_component('nanpy', GenericNanpy, ARDUINO_PORT)
        self.add_router(bind_to='tcp://*:5555')

MyApplication().run()
```

6.2.3 A Client for our Arkady Application

It should be noted, that because Arkady makes use of ZeroMQ, a client can be written in nearly any language as bindings are widely implemented. The following example is simply an example in Python using PyZMQ. It is interactive so that you might test out sending arbitrary messages to your Arduino (via Nanpy, via Arkady!).

```
#!/usr/bin/env python3

"""
Demonstration of a program controlling the very generic Arkady interface to
NanPy.

Direct dependencies are: pyzmq
"""

import zmq

NANPY_ADRRESS = 'tcp://localhost:5555'  # replace localhost with IP if remote

context = zmq.Context()
sock = context.socket(zmq.REQ)
sock.connect(NANPY_ADRRESS)

while True:
    msg = input('Send a message to the Nanpy device: ')
    # set first word as "nanpy" so message goes to our registered component
    sock.send_string('nanpy ' + msg)
    print('Waiting for reply.')
    reply = sock.recv_string()
    print('Got: ' + reply)
```

6.2.4 A Quick Guide to Setting up Nanpy

Using Nanpy from your computer is as simple as:

```
pip install nanpy
```

To put the corresponding firmware on your Arduino, you can get a copy of this firmware with the following command. These instructions are also outlined there.

```
git clone https://github.com/nanpy/nanpy-firmware.git
```

Then change directories to the subsequent `nanpy-firmware` directory and execute `./configure.sh`. Then copy the `nanpy-firmware/Nanpy` directory into your Arduino sketchbook directory.

Plug in your Arduino, start the Arduino IDE, configure your boardset and port, open the Nanpy module from the sketchbook, and then Upload. Assuming everything went well, your Arduino should be ready for Nanpy control.

6.3 Components

A Component 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: *SerialComponent* and *AsyncComponent*. Use of *SerialComponent* is recommended when the underlying work must be strictly serial (meaning non-parallel). *AsyncComponent* is suitable when multiple executions of the *handler* can safely run simultaneously.

class arkady.components.**AsyncComponent**(**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.components.**Component**(**args*, *loop=None*, ***kwargs*)
The Base Component 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.components.**DummyAsyncDevice**(**args*, ***kwargs*)

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

class arkady.components.**SerialComponent**(**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

6.4 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