Tubes Documentation

Release 0.0.0

Twisted Matrix Labs

May 23, 2017

Contents

Contents:

CHAPTER 1

An Introduction to Tubes

What Are Tubes?

The tubes package provides composable flow-control and data processing.

Flow-control is control over the source, destination, and rate of data being processed. Tubes implements this in a type-agnostic way, meaning that a set of rules for controlling the flow of data can control that flow regardless of the type of that data, from raw streams of bytes to application-specific messages and back again.

Composable data processing refers to processing that can occur in independent units. For example, the conversion of a continuous stream of bytes into a discrete sequence of messages can be implemented independently from the presentation of or reactions to those messages. This allows for similar messages to be relayed in different formats and by different protocols, but be processed by the same code.

In this document, you will learn how to compose founts (places where data comes from), drains (places where data goes to), and tubes (things that modify data by converting inputs to outputs) to create flows. You'll also learn how to create your own tubes to perform your own conversions of input to output. By the end, you should be able to put a series of tubes onto the Internet as a server or a client.

Getting Connected: an Echo Server

Let's start with an example. The simplest way to process any data is to avoid processing it entirely, to pass input straight on to output. On a network, that means an echo server as described in **RFC 862**. Here's a function which uses interfaces defined by `tubes` to send its input straight on to its output:

```
def echoFlow(flow):
    flow.fount.flowTo(flow.drain)
```

In the above example, `echoFlow` takes two things: a fount, or a source of data, and a drain , or a place where data eventually goes. Such a function is called a "flow", because it establishes a flow of data from one place to another. Most often, the arguments to such a function are the input from and the output to the same network connection. The fount represents data coming in over the connection, and the drain represents data going back out over that same connection.

To *use* `echoFlow` as a server, you have to attach it to a listening endpoint.

`echoflow.py`

```
from tubes.protocol import flowFountFromEndpoint
from tubes.listening import Listener

from twisted.internet.endpoints import serverFromString
from twisted.internet.defer import Deferred, inlineCallbacks

def echoFlow(flow):
    flow.fount.flowTo(flow.drain)

@inlineCallbacks
def main(reactor, listenOn="stdio:"):
    endpoint = serverFromString(reactor, listenOn)
    flowFount = yield flowFountFromEndpoint(endpoint)
    flowFount.flowTo(Listener(echoFlow))
    yield Deferred()

if __name__ == '__main__':
    from twisted.internet.task import react
    from sys import argv
    react(main, argv[1:])
```

This fully-functioning example (just run it with "`python echoflow.py`") implements an echo server. By default, you can test it out by typing into it.

```
$ python echoflow.py
are you an echo server?
are you an echo server?
^C
```

If you want to see this run on a network, you can give it an endpoint description. For example, to run on TCP port 4321:

```
$ python echoflow.py tcp:4321
```

and then in another command-line window:

```
$ telnet 127.0.0.1 4321
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
are you an echo server?
are you an echo server?
^]
telnet> close
Connection closed.
```

You can test it out with `telnet localhost 4321`.

Note: If you are on Windows, `telnet` is not installed by default. If you see an error message like:

```
'telnet' is not recognized as an internal or external command,
operable program or batch file.
```

then you can install `telnet` by running the command

```
C:\> dism /online /Enable-Feature /FeatureName:TelnetClient
```

in an Administrator command-prompt first.

However, this example still performs no processing of the data that it is receiving.

Processing A Little Data: Reversing A String

Let's perform some very simple processing on our input data: we will reverse each line that we receive.

This immediately raises the question: how do we tell when we have received a whole line? The previous echo example didn't care because it would just emit whatever bytes were sent to it, regardless of whether it had received a whole message or not. (It just so happens that your terminal only sends the bytes when you hit the "enter" key.) We can't just split up the incoming data with `bytes.split` because we might receive one line, part of a line, or multiple lines in one network message. Luckily Tubes implements this for us, with the handy tubes.framing module (so called because it puts "frames" around chunks of bytes, and you can distinguish one chunk from the next).

Note: There are many types of framing mechanisms, and the one we're demonstrating here, line-oriented message separation, while it is extremely common, is one of the worst ones. For example, a line-delimited message obviously cannot include a newline, and if you try to transmit one that does, you may get a garbled data stream. The main advantage of a line-separated protocol is that it works well for interactive examples, since a human being can type lines into a terminal. For example, it works well for documentation :-). However, if you're designing your own network protocol, please consider using a length-prefixed framing mechanism, such as netstrings.

Much like in the echo example, we need a flow function which sets up the flow of data from a fount to a drain.

```
def reverseFlow(flow):
    from tubes.framing import bytesToLines, linesToBytes
    lineReverser = series(bytesToLines(), Reverser(), linesToBytes())
    flow.fount.flowTo(lineReverser).flowTo(flow.drain)
```

In this flow function, we have a new object, a *series of Tubes*, created by the series function. You can read the construction of the `lineReverser` series as a flow of data from left to right. The output from each tube in the series is passed as the input to the tube to its right.

We are expecting a stream of bytes as our input, because that's the only thing that ever comes in from a network. Therefore the first element in the series, bytesToLines, as its name implies, converts the stream of bytes into a sequence of lines. The next element in the series, `Reverser`, reverses its inputs, which, being the output of bytesToLines, are lines.

`Reverser` is implemented like so:

```
class Reverser(object):
    def received(self, item):
        yield b"".join(reversed(item))
```

Managing State with a Tube: A Networked Calculator

To demonstrate both receiving and processing data, let's write a reverse Polish notation calculator for addition and multiplication.

Interacting with it should look like this:

```
3
4
+
7
2
*
14
```

In order to implement this program, you will construct a *series* of objects which process the data; specifically, you will create a series of Tubes. Each tubes.Tube in the tubes.series will be responsible for processing part of the data.

Lets get started with just the core component that will actually perform calculations.

```
class Calculator(object):
    def __init__(self):
        self.stack = []

    def push(self, number):
        self.stack.append(number)

    def do(self, operator):
        left = self.stack.pop()
        right = self.stack.pop()
        result = operator(left, right)
        self.push(result)
        return result
```

`Calculator` gives you an API for pushing numbers onto a stack, and for performing an operation on the top two items in the stack, the result of which is then pushed to the top of the stack.

Now let's look at the full flow which will pass inputs to a `Calculator` and relay its output:

```
def calculatorSeries():
    from tubes.tube import series
    from tubes.framing import bytesToLines, linesToBytes

    return series(
        bytesToLines(),
        linesToNumbersOrOperators,
        CalculatingTube(Calculator()),
        numbersToLines,
        linesToBytes()
    )
```

The first tube in this series, provided by the tubes.framing module, transforms a stream of bytes into lines. Then, `linesToNumbersOrOperators` - which you'll write in a moment - should transform lines into a combination of numbers and operators (functions that perform the work of the `"+"` and `"*"` commands), then from numbers and operators into more numbers - sums and products - from those integers into lines, and finally from those lines into newline-terminated segments of data that are sent back out. A `CalculatingTube` should pass those numbers and operators to a `Calculator`, and produce numbers as output. `numbersToLines` should convert the output numbers into byte strings, and *linesToBytes* performs the inverse of *bytesToLines* by appending newlines to those outputs.

Let's look at `linesToNumbersOrOperators`.

```
@tube
def linesToNumbersOrOperators(line):
    from operator import add, mul
    try:
        yield int(line)
    except ValueError:
        if line == '+':
            yield add
        elif line == '*':
            yield mul
```

ITube.received takes an input and produces an iterable of outputs. A tube's input is the output of the tube preceding it in the series. In this case, `linesToNumbersOrOperators` receives the output of bytesToLines, which outputs sequences of bytes (without a trailing line separator). Given the specification for the RPN calculator's input above, those lines may contain ASCII integers (like `b"123"`) or ASCII characters representing arithmetic operations (`b"+"` or `b"*"`). `linesToNumbersOrOperators` output falls into two categories: each line containing decimal numbers results in an integer output, and each operator character is represented by a python function object that can perform that operation.

Now that you've parsed those inputs into meaningful values, you can send them on to the `Calculator` for processing.

```
@tube
class CalculatingTube(object):
    def __init__(self, calculator):
        self.calculator = calculator

    def received(self, value):
        if isinstance(value, int):
            self.calculator.push(value)
        else:
            yield self.calculator.do(value)
```

`CalculatingTube` takes a `Calculator` to its constructor, and provides a *received* method which takes, as input, the outputs produced by *LinesToNumbersOrOperators*. It needs to distinguish between the two types it might be handling — integers, or operators — and it does so with *isinstance*. When it is handling an integer, it pushes that value onto its calculator's stack, and, importantly, does not produce any output. When it is handling an operator, it applies that operator with its calculator's *do* method, and outputs the result (which will be an integer).

Unlike `linesToNumbersOrOperators`, `CalculatingTube` is *stateful*. It does not produce an output for every input. It only produces output when it encounters an operator.

Finally we need to move this output along so that the user can see it.

To do this, we use the very simple `numbersToLines` which takes integer inputs and transforms them into ASCII bytes.

```
@tube
def numbersToLines(value):
    yield str(value).encode("ascii")
```

Like `linesToNumbersOrOperators`, `numbersToLines` is stateless, and produces one output for every input.

Before sending the output back to the user, you need to add a newline to each number so it is legible to the user. Otherwise the distinct numbers "3", "4", and "5" would show up as "345".

For this, we use the aforementioned `bytesToLines` tube, which appends newlines to its inputs.

CHAPTER 2

Tubes Versus Protocols

If you've used Twisted before, you may notice that half of the line-splitting above is exactly what LineReceiver does, and that there are lots of related classes that can do similar things for other message types. The other half is handled by producers and consumers. `tubes` is a *newer* interface than those things, and you will find it somewhat improved. If you're writing new code, you should generally prefer to use `tubes`.

There are three ways in which `tubes` is better than using producers, consumers, and the various `XXXReceiver` classes directly.

1. `tubes` is *general purpose*. Whereas each `FooReceiver` class receives `Foo` objects in its own way, `tubes` provides consistent, re-usable abstractions for sending and receiving.
2. `tubes` *does not require subclassing*. The fact that different responsibilities live in different objects makes it easier to test and instrument them.
3. `tubes` *handles flow-control automatically*. The manual flow-control notifications provided by `IProducer` and `IConsumer` are still used internally in `tubes` to hook up to `twisted.internet`, but the interfaces defined in `tubes` itself are considerably more flexible, as they allow you to hook together chains of arbitrary length, as opposed to just getting buffer notifications for a single connection to a single object.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index