Nanostream Documentation

Release 0.1

Zachary Ernst

Oct 12, 2018

CONTENTS:

CHAPTER

ONE

OVERVIEW

1.1 What is it? Why is it?

We love stream processing. It's a great model for lots of work, especially ETL. There are excellent stream processing tools such as Spark, Flink, and Storm. They're designed for handling huge amounts of Big Data(tm). But they carry a lot of overhead because they're Big Data(tm) tools.

Most of our data problems are not Big Data(tm). They're also not Small Data. They're Medium Data – that is, data that's big enough to require some planning, but not so big as to justify the infrastructure and complexity overhead that come with Spark and its cousins.

NanoStream lets you deploy a streaming application with no overhead. It's entirely self-contained, and runs on a single core (which, let's face is, is more than enough processing power for 99% of your work). NanoStream sets up each step in your pipeline in its own thread, so there are no bottlenecks. It monitors all the threads and queues, and logs any problems. If the data comes in faster than NanoStream can handle, it applies back-pressure to the data stream. But in reality, because NanoStream doesn't have any of the overhead of distributed systems, it's pretty fast.

1.1.1 Using NanoStream

You use NanoStream by specifying one or more `NanoNode` objects, linking them together into a pipeline (an acyclic directed graph), and starting them. Several `NanoNode` classes are provided, and it's easy to create new ones. Here are some types of examples:

1.2 Using built-in `NanoNode` classes

Let's say you want to watch a directory for new CSV files, read them when they appear, iterate over all the rows, and print those rows as they arrive. You can do so by importing a few classes, instantiating them, and running them in a pipeline like so:

```
# Instantiate the classes:
watchdog = LocalDirectoryWatchdog(directory='./data_directory')
file_reader = LocalFileReader(serialize=False)
csv_reader = CSVReader()
printer = PrinterOfThings()

# Use ">" to create connections between the nodes
watchdog > file_reader > csv_reader > printer

# Start it
# You can run the `global_start` method on any of the connected nodes;
```

(continues on next page)

(continued from previous page)

```
# it will automatically start all of them.
watchdog.global_start()
```

The result will be a streaming pipeline that monitors `data_directory/`, printing the rows of any CSV file that appears there (or is modified).

1.3 Rolling your own `NanoNode` class

`NanoNode` objects fall into one of two categories, depending on whether they ingest data from other nodes, or generate data another way. If they accept data from an upstream `NanoNode`, then you specify a `process_item` method; if they generate their own data (i.e. they're at the beginning of the pipeline), then you specify a `generator` method. Your class should inherit from `NanoNode`, and you provide the appropriate method (`process_item` or `generator`), and if necessary, define an `__init__` method.

For example, suppose you want to create a source node for your pipeline that simply emits the word `foo` every few seconds, and the user specifies how many seconds between each `foo`. Then the class would be defined like so:

```
class FooEmitter(NanoNode):  # inherit from NanoNode
    def __init__(self, message='', interval=1):
        self.message = message
        self.interval = interval
        super(FooEmitter, self).__init__()  # Must call the `NanoNode` __init__

    def generator(self):
        while self.run_generator():
            time.sleep(self.interval)
            yield message  # Output must be yielded, not returned
```

Of course, the example is trivial because you generally won't want to keep sending the same string over and over again forever. More realistic uses of this pattern would include reading lines from a file, connecting to an external API, and so on.

Note that the code inside the `generator` function is wrapped inside a `while` loop that tests the value of `self.run_generator()`. You'll want to ensure that your generator methods follow the same pattern because

Now let's suppose you want to create a node that is passed a string as a message, and returns `True` if the message has an even number of characters, `False` otherwise. The class definition would look like this:

```
class MessageLengthTester(NanoNode):
    def __init__(self):
        # No particular initialization required in this example
        super(MessageLengthTester, self).__init__()

    def process_item(self, message):
        if len(message) % 2 == 0:
            yield True  # Again, note the use of yield instead of return
        else:
            yield False
```

That's it.

Instantiating both of them into a pipeline is just a matter of instantiating the classes and hooking them together:

```
message_node = FooEmitter(message='foobar', interval=5)
length_tester_node = MessageLengthTester()
```

(continues on next page)

(continued from previous page)

```
message_node > length_tester_node

message_node.global_start()
```

1.4 Composing and configuring NanoNode objects

Let's suppose you've worked very hard to create the pipeline from the last example. Now, your boss says that another engineering team wants to use it, but they want to rename parameters and "freeze" the values of certain other parameters to specific values. Once that's done, they want to use it as just one part of a more complicated NanoStream pipeline.

This can be accomplished using a configuration file. When NanoStream parses the configuration file, it will dynamically create the desired class, which can be instantiated and used as if it were a single node in another pipeline.

The configuration file is written in YAML, and it would look like this:

```
name: FooMessageTester

nodes:
  - name: foo_generator
    class FooEmitter
    frozen_arguments:
      message: foobar
    arg_mapping:
      interval: foo_interval
  - name: length_tester
    class: MessageLengthTester
    arg_mapping: null
```

With this file saved as (e.g.) foo_message.yaml, the following code will create a FooMessageTester class and instantiate it:

```
foo_message_config = yaml.load(open('./foo_message.yaml', 'r').read())
class_factory(foo_message_config)
# At this point, there is now a `FooMessageTester` class
foo_node = FooMessageTester(foo_interval=1)
```

You can now use foo_node just as you would any other node. So in order to run it, you just do:

```
foo_node.global_start()
```

Because foo_node is just another node, you can insert it into a larger pipeline and reuse it. For example, suppose that other engineering team wants to add a PrinterOfThings to the end of the pipeline. They'd do this:

```
printer = PrinterOfThings()
foo_node > printer
```

CHAPTER

TWO

REQUIREMENTS

NanoStream is written in Python 3.5. All requirements are pip-installable and are listed n the `requirements.txt` file:

```
` pip install -r requirements.txt `
```

The documentation is written using Sphinx, so if you want to rebuild the docs, you'll do:

```
` make [html | latexpdf | whatever] `
```

That ought to be everything.

CHAPTER

THREE

IMPLEMENTATION

Nothing here yet.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search