PyProcSync Documentation

Release 0.1.0

Marcell Pünkösd

Apr 07, 2021

CONTENTS:

CHAPTER

ONE

PYPROCSYNC

Synchronize events between processes over the network. This package provides similar behaviour as Python's *threading.Event* but it is designed to be used with multiple processes running on different computers.

An example use-case might be controlling multiple industrial robots handling anomalous materials, where timing is critical.

- Free software: MIT license
- Documentation: https://pyprocsync.readthedocs.io

1.1 Features

- Uses Redis as a backend
- About 1ms precision (see. *perf_tests*)
- Synchronize events based on system clock (NTP is a must have)
- Synchronize unlimited number of nodes with the same precision (Depends on the performance of Redis cluster)

1.2 Example

Simple example that synchronizes 4 nodes:

```
import redis
from pyprocsync import ProcSync

p = ProcSync(redis.from_url('redis://localhost:6379/0'))

# Do some work

p.sync("first", 4) # Block until all 4 nodes are reached the synchronization point

# Time sensitive work
```

1.3 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER

TWO

INSTALLATION

2.1 Stable release

To install PyProcSync, run this command in your terminal:

```
$ pip install pyprocsync
```

This is the preferred method to install PyProcSync, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for PyProcSync can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/marcsello/pyprocsync
```

Or download the tarball:

```
$ curl -OJL https://github.com/marcsello/pyprocsync/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER

THREE

USAGE

3.1 Basics

To use PyProcSync in a project:

```
from pyprocsync import ProcSync
```

In order to work you need a Redis server (or a cluster in large deployments) accessible by all the nodes that will run code that needs to be synchronized:

```
import redis
procsync = ProcSync(redis.from_url("redis://localhost:6379/0"), "testrun1")
```

IMPORTANT: The *run_id* (testrun1 in the above example) must be unique and the same on all nodes. After running a synchronized scenario the run_id must be changed to something different (and set the same on all nodes).

When your program reaches a point where it needs to synchronize to other nodes before continuing execution simply call *sync* and supply the name of the synchronization point and the nodes needed to be synchronized:

```
# doing some work asychronously

procsync.sync("test1", 3)  # <- Wait for all 3 nodes to reach the synchronization␣
↪point named "test1"

# continue executing at the same time as the other three nodes
```

NOTE: You do not need all your nodes that takes part of the same run to call *sync* thus you don't need to set the nodes parameter to the number of all nodes. This allows you to define synchronization points only a portion of nodes need to synchronize to. But always make sure that the right amount of nodes are specified otherwise you may face unforeseen consequences.

You may call *close* after you no longer need to synchronize or your scenario reached the end. This unsubscribes from the subscribed pubsub channel, and closes Redis connection:

```
procync.close()
```

3.2 Timeout

It is possible to set a timeout when waiting for other nodes to synchronize. In an ideal world this would be unnecessary because it's purpose is to allow developers recover from a scenario where one or more of the nodes became offline or simply taking an irrational amount of time to synchronize.

By default timeouts are disabled so each node waits infinite amount of time for a synchronized condition.

Using a timeout is simple, provide it as the third parameter to the *sync* call (in seconds):

```
from pyprocsync import TimeOutError

try:
    procsync.sync("test2", 3, 60)
except TimeOutError:
    # The synchronization could not be done in a minute.
    # Terminate everything!
```

3.3 Setting the delay

Delay is the amount of time (seconds) added to the current time when announced. In other words it's the amount of time you give each node to receive the announced time of continue and wait the remaining time in order to continue the execution at the exact same time on each node.

Based on your network you may want to change this delay. The default value is 1 sec.

For example...

- If your nodes are on remote locations where latency may go above 1 sec, and you encounter ANY *TooLateError* you HAVE TO increase the delay.
- On the other hand, if all your processes run very close to eachother (even on the same host) you can decrease the delay to make synchronizations happen faster.

The delay is assigned when you create the ProcSync object:

```
from pyprocsync import ProcSync, TooLateError

procsync = ProcSync(redis.from_url("redis://localhost:6379/0"), "testrun2", 10)

try:
    procsync.sync("test2", 3)
except TooLateError:
    # 10 seconds wasn't enough time to distribute the time of continue
    # There is some serious problem
    # Terminate everything!
```

NOTE: When the delay is low enough, slight drifts between system clocks can also result in *TooLateError* error.

CHAPTER

FOUR

WORKING PRINCIPLE

PyProcSync's magic depends on important features of Redis: The atomic increment-and-get and the built in pubsub message broker.

4.1 The idea

The goal of PyProcSync is to solve time critical program synchronization over the network between multiple nodes. One simple and obvious solution is to use the nodes' clock since those can be synchronized very precisely using NTP and PTP. This way we don't need complicated and exotic synchronization protocols.

The basic idea of PyProcSync is that each node may agree on a future timestamp on which they continue the execution. In theory this eliminates the network latency in the system completely.

This "agreement" protocol is realized using a Redis as shared memory and message broker.

4.2 Deciding the time of continue

Given that we know of how many nodes are required to synchronize their execution. We simply count if all nodes reached the synchronization point using a simple counter incremented when a process reached the synchronization point (conceptually similar to semaphores).

The last node reaching the synchronization point can identify that all nodes reached the point because the value of the counter will be equal to the number of the total number of required nodes. Then the last node uses it's own system clock, adds a predefined amount of time and then publish that future timestamp.

Each node is then receive the timestamp calculated by the last node. And uses their own system clock to calculate the amount time it should sleep before continuing the execution.

4.3 Flow

The following figure depicts the basic flow of a synchronization using PyProcSync.

CHAPTER

FIVE

MODULE OVERVIEW

Top-level package for PyProcSync.

5.1 Classes

5.1.1 PyProcSync

class pyprocsync.pyprocsync.**ProcSync**(*redis_client: redis.client.Redis*, *run_id: str = ''*, *delay: float = 1.0*)

Bases: object

PyProcSync main class.

An instance of class represents a single "run" with their own Redis connection. Each synchronization point (*sync()* calls) is tied to the context of a ProcSync instance with an unique *run_id*.

Upon creation the appropriate channel is being subscribed to.

Although the *run_id* is optional it is strongly recommended to be set to a different value at each creation. The *run_id* must be the same on all nodes that takes part of the same "run".

Parameters

- **redis_client** – A *redis.Redis* instance that's connected to a redis server.
- **run_id** – An arbitrary id (str) that identifies this specific run. (Should be unique across all runs and the same on all nodes)
- **delay** – Time spent waiting after the continue time is announced. (Default is 1 sec)

close()

Close Redis connection. After calling this method. The instance should not be used anymore.

sync(*event_name: str*, *nodes: int*, *timeout: Optional[float] = None*)

Start waiting for each node (number of nodes specified by *nodes* parameter) to arrive at the synchronization point specified by *event_name*.

WARNING: All parameters of this method MUST BE the same on every node for the same event (including timeout). If parameters supplied for this method differ from other nodes, this would not only cause malfunction in the current instance but would confuse other nodes waiting for this synchronization point as well!

If configured properly, this method returns at the same time (according to their system clock) on all nodes.

Exceptions this method may raise:

- ValueError: Some parameters are invalid.

- pyprocsync.TooLateError: Synchronization time already expired when recieved (system clocks not in sync or configured delay lower than network latency)
- pyprocsync.TimeOutError: (only when timeout is not none) Given up waiting for other nodes.
- AssertionError: Unexpected values read from Redis.
- Redis related exceptions (see. pyredis docs).

Parameters

- **event_name** – The name of the event. This is the same across all nodes that want to synchronize.
- **nodes** – Amount of nodes to sync the event between.
- **timeout** – Maximum time to wait for all nodes to reach the synchronization point defined by event_name in seconds. Set to *None* for infinite wait time. TimeOutError raised when the timeout expire.

5.1.2 Exceptions

This module contains all custom exceptions raised by PyProcSync.

exception pyprocsync.exceptions.**ProcSyncError**
Bases: BaseException

Base class for all exceptions in PyProcSync.

exception pyprocsync.exceptions.**TimeOutError**
Bases: *pyprocsync.exceptions.ProcSyncError*

This exception is raised when the node is gave up waiting for other nodes. This could caused by other nodes crashed or bad configuration.

exception pyprocsync.exceptions.**TooLateError**
Bases: *pyprocsync.exceptions.ProcSyncError*

This exception is raised when the announced continue time is already passed. That could be caused by high network latency or unsynchronized system clocks between nodes.

CHAPTER

SIX

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/marcsello/pyprocsync/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

6.1.4 Write Documentation

PyProcSync could always use more documentation, whether as part of the official PyProcSync docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/marcsello/pyprocsync/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *pyprocsync* for local development.

1. Fork the *pyprocsync* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pyprocsync.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv pyprocsync
$ cd pyprocsync/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 pyprocsync tests
$ python setup.py test or pytest
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.5, 3.6, 3.7 and 3.8, and for PyPy. Check https://travis-ci.com/marcsello/pyprocsync/pull_requests and make sure that the tests pass for all supported Python versions.

6.4 Tips

To run a subset of tests:

```
$ pytest tests.test_pyprocsync
```

6.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bump2version patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER

SEVEN

CREDITS

7.1 Development Lead

- Marcell Pünkösd <punkosdmarcell@rocketmail.com>

7.2 Contributors

None yet. Why not be the first?

CHAPTER

EIGHT

HISTORY

8.1 0.1.0 (2021-04-05)

- First release on PyPI.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX