<PROJECT$_{N}AME > Documentation$

Release 0.12.0

Jason Carver

May 08, 2019

General

CHAPTER 1

Lahja

Warning: This is a very young project. It's used and validated mainly by the Python Ethereum client (Trinity) Lahja is alpha state software. Expect bugs.

Lahja is a generic multi process event bus implementation written in Python 3.6+ that enables lightweight inter-process communication, based on non-blocking asyncio.

1.1 Goals

Lahja is tailored around one primary use case: Enabling event-based communication between different processes in moder Python applications using non-blocking asyncio.

Features:

- Non-blocking APIs based on `asyncio`
- Broadcast events within a single process or across multiple processes.
- Multiple APIs to consume events that adapt to different use cases and styles
- lightweight and simple (e.g. no IPC pipe management etc)
- Easy event routing (e.g. route specific events to specific processes or process groups)

1.2 Further reading

Here are a couple more useful links to check out.

- Source Code on GitHub
- Examples

CHAPTER 2

Table of contents

2.1 Introduction

2.1.1 Lahja

Warning: This is a very young project. It's used and validated mainly by the Python Ethereum client (Trinity) Lahja is alpha state software. Expect bugs.

Lahja is a generic multi process event bus implementation written in Python 3.6+ that enables lightweight inter-process communication, based on non-blocking asyncio.

Goals

Lahja is tailored around one primary use case: Enabling event-based communication between different processes in moder Python applications using non-blocking asyncio.

Features:

- Non-blocking APIs based on `asyncio`
- Broadcast events within a single process or across multiple processes.
- Multiple APIs to consume events that adapt to different use cases and styles
- lightweight and simple (e.g. no IPC pipe management etc)
- Easy event routing (e.g. route specific events to specific processes or process groups)

Further reading

Here are a couple more useful links to check out.

- Source Code on GitHub
- Examples

2.2 Quickstart

2.2.1 Install the library

```
pip install lahja
```

2.2.2 Import Endpoint and BaseEvent

```
import asyncio
import multiprocessing
import time

from lahja import (
    BaseEvent,
    Endpoint,
    ConnectionConfig,
)
```

2.2.3 Setup application specific events

```
class BaseExampleEvent(BaseEvent):

    def __init__(self, payload):
        super().__init__()
        self.payload = payload
```

```
class FirstThingHappened(BaseExampleEvent):
    pass
```

```
class SecondThingHappened(BaseExampleEvent):
    pass
```

2.2.4 Setup first process to receive and broadcast events

```
def run_proc1():
    loop = asyncio.get_event_loop()
    loop.run_until_complete(proc1_worker(endpoint))
```

```
async def proc1_worker(endpoint):
    endpoint = Endpoint()
    await endpoint.start_serving(ConnectionConfig.from_name('e1'))
    await endpoint.connect_to_endpoints(
```

(continues on next page)

(continued from previous page)

```
        ConnectionConfig.from_name('e2')
    )
    endpoint.subscribe(SecondThingHappened, lambda event:
        print("Received via SUBSCRIBE API in proc1: ", event.payload)
    )
    endpoint.subscribe(FirstThingHappened, lambda event:
        print("Receiving own event: ", event.payload)
    )

    while True:
        print("Hello from proc1")
        if is_nth_second(5):
            await endpoint.broadcast(
                FirstThingHappened("Hit from proc1 ({})".format(time.time()))
            )
        await asyncio.sleep(1)
```

2.2.5 Setup second process to receive and broadcast events

```
def run_proc2():
    loop = asyncio.get_event_loop()
    loop.run_until_complete(proc2_worker())
```

```
async def proc2_worker():
    endpoint = Endpoint()
    await endpoint.start_serving(ConnectionConfig.from_name('e2'))
    await endpoint.connect_to_endpoints(
        ConnectionConfig.from_name('e1')
    )
    asyncio.ensure_future(display_proc1_events(endpoint))
    endpoint.subscribe(FirstThingHappened, lambda event:
        print("Received via SUBSCRIBE API in proc2:", event.payload)
    )
    while True:
        print("Hello from proc2")
        if is_nth_second(2):
            await endpoint.broadcast(
                SecondThingHappened("Hit from proc2 ({})".format(time.time()))
            )
        await asyncio.sleep(1)
```

2.2.6 Start both processes

```
    p1 = multiprocessing.Process(target=run_proc1)
    p1.start()

    p2 = multiprocessing.Process(target=run_proc2)
    p2.start()
```

2.3 Running the examples

2.3.1 Example: Chatter between two processes

```
python examples/inter_process_ping_pong.py
```

The output will look like this:

```
Hello from proc1
Hello from proc2
Received via SUBSCRIBE API in proc1:  Hit from proc2 (1533887068.9261594)
Hello from proc1
Hello from proc2
Hello from proc1
Hello from proc2
Received via SUBSCRIBE API in proc1:  Hit from proc2 (1533887070.9296985)
Receiving own event:  Hit from proc1 (1533887070.9288142)
Received via SUBSCRIBE API in proc2: Hit from proc1 (1533887070.9288142)
Received via STREAM API in proc2:  Hit from proc1 (1533887070.9288142)
Hello from proc1
Hello from proc2
Hello from proc1
Hello from proc2
Received via SUBSCRIBE API in proc1:  Hit from proc2 (1533887072.9331954)
Hello from proc1
Hello from proc2
Hello from proc1
Hello from proc2
Received via SUBSCRIBE API in proc1:  Hit from proc2 (1533887074.937018)
Hello from proc1
Hello from proc2
Received via SUBSCRIBE API in proc2: Hit from proc1 (1533887075.9378386)
Received via STREAM API in proc2:  Hit from proc1 (1533887075.9378386)
Receiving own event:  Hit from proc1 (1533887075.9378386)
```

2.3.2 Example: Request API

```
python examples/request_api.py
```

The output will look like this:

```
Requesting
Got answer: Yay
Requesting
Got answer: Yay
Requesting
Got answer: Yay
```

2.4 API Docs

2.4.1 Endpoint

class lahja.endpoint.**Broadcast**(*event*, *config*)

Bases: `tuple`

config

Alias for field number 1

event

Alias for field number 0

class lahja.endpoint.**ConnectionConfig**

Bases: `tuple`

Configuration class needed to establish *Endpoint* connections.

classmethod from_name(*name: str*, *base_path: Optional[pathlib.Path] = None*) → lahja.endpoint.ConnectionConfig

name

Alias for field number 0

path

Alias for field number 1

class lahja.endpoint.**Endpoint**

Bases: `object`

The *Endpoint* enables communication between different processes as well as within a single process via various event-driven APIs.

TResponse = ~TResponse

TStreamEvent = ~TStreamEvent

TSubscribeEvent = ~TSubscribeEvent

TWaitForEvent = ~TWaitForEvent

broadcast(*item: lahja.misc.BaseEvent*, *config: Optional[lahja.misc.BroadcastConfig] = None*) → None

Broadcast an instance of *BaseEvent* on the event bus. Takes an optional second parameter of *BroadcastConfig* to decide where this event should be broadcasted to. By default, events are broadcasted across all connected endpoints with their consuming call sites.

broadcast_nowait(*item: lahja.misc.BaseEvent*, *config: Optional[lahja.misc.BroadcastConfig] = None*) → None

A non-async *broadcast()* (see the docstring for *broadcast()* for more)

Instead of blocking the calling coroutine this function schedules the broadcast and immediately returns.

CAUTION: You probably don't want to use this. broadcast() doesn't return until the write socket has finished draining, meaning that the OS has accepted the message. This prevents us from sending more data than the remote process can handle. broadcast_nowait has no such backpressure. Even after the remote process stops accepting new messages this function will continue to accept them, which in the worst case could lead to runaway memory usage.

check_event_loop() → TFunc

All Endpoint methods must be called from the same event loop.

connect_to_endpoint(*config: lahja.endpoint.ConnectionConfig*) → None

connect_to_endpoints (**endpoints*) → None
Connect to the given endpoints and await until all connections are established.

connect_to_endpoints_nowait (**endpoints*) → None
Connect to the given endpoints as soon as they become available but do not block.

event_loop

has_snappy_support

ipc_path

is_connected_to (*endpoint_name: str*) → bool

logger

name

request (*item: lahja.misc.BaseRequestResponseEvent[TResponse], config: Optional[lahja.misc.BroadcastConfig] = None*) → TResponse
Broadcast an instance of `BaseRequestResponseEvent` on the event bus and immediately wait on an expected answer of type `BaseEvent`. Optionally pass a second parameter of `BroadcastConfig` to decide where the request should be broadcasted to. By default, requests are broadcasted across all connected endpoints with their consuming call sites.

start_serving (*connection_config: lahja.endpoint.ConnectionConfig*) → None
Start serving this `Endpoint` so that it can receive events. Await until the `Endpoint` is ready.

stop () → None
Stop the `Endpoint` from receiving further events.

stream (*event_type: Type[TStreamEvent], num_events: Optional[int] = None*) → AsyncGenerator[TStreamEvent, None]
Stream all events that match the specified event type. This returns an `AsyncIterable[BaseEvent]` which can be consumed through an `async for` loop. An optional `num_events` parameter can be passed to stop streaming after a maximum amount of events was received.

subscribe (*event_type: Type[TSubscribeEvent], handler: Callable[TSubscribeEvent, None]*) → lahja.misc.Subscription
Subscribe to receive updates for any event that matches the specified event type. A handler is passed as a second argument an `Subscription` is returned to unsubscribe from the event if needed.

wait_for (*event_type: Type[TWaitForEvent]*) → TWaitForEvent
Wait for a single instance of an event that matches the specified event type.

wait_until_serving () → None
Await until the `Endpoint` is ready to receive events.

class lahja.endpoint.**RemoteEndpoint** (*reader: asyncio.streams.StreamReader, writer: asyncio.streams.StreamWriter*)
Bases: `object`

static connect_to (*path: str*) → lahja.endpoint.RemoteEndpoint

read_message () → lahja.endpoint.Broadcast

send_message (*message: lahja.endpoint.Broadcast*) → None

2.4.2 ConnectionConfig

class lahja.endpoint.**ConnectionConfig**
Bases: `tuple`

Configuration class needed to establish `Endpoint` connections.

classmethod from_name(*name: str*, *base_path: Optional[pathlib.Path] = None*) → lahja.endpoint.ConnectionConfig

name
Alias for field number 0

path
Alias for field number 1

2.4.3 Exceptions

exception lahja.exceptions.**ConnectionAttemptRejected**
Bases: `Exception`

Raised when an attempt was made to connect to an endpoint that is already connected.

exception lahja.exceptions.**LahjaError**
Bases: `Exception`

Base class for all lahja errors

exception lahja.exceptions.**NotServing**
Bases: `lahja.exceptions.LahjaError`

Raised when an API call was made before `start_serving()` was called.

exception lahja.exceptions.**ReaderAtEof**
Bases: `lahja.exceptions.LahjaError`

Raised by RemoteEndpoint when the remote has closed the connection.

exception lahja.exceptions.**UnexpectedResponse**
Bases: `lahja.exceptions.LahjaError`

Raised when the type of a response did not match the `expected_response_type`.

2.4.4 BaseEvent

class lahja.misc.**BaseEvent**
Bases: `object`

broadcast_config(*internal: bool = False*) → lahja.misc.BroadcastConfig

2.4.5 BaseRequestResponseEvent

class lahja.misc.**BaseRequestResponseEvent**
Bases: `abc.ABC`, `lahja.misc.BaseEvent`, `typing.Generic`

static expected_response_type() → Type[TResponse]
Return the type that is expected to be send back for this request. This ensures that at runtime, only expected responses can be send back to callsites that issued a *BaseRequestResponseEvent*

2.4.6 BroadcastConfig

class lahja.misc.**BroadcastConfig**(*filter_endpoint: Optional[str] = None*, *filter_event_id: Optional[str] = None*, *internal: bool = False*)

Bases: object

allowed_to_receive(*endpoint: str*) → bool

2.4.7 Subscription

class lahja.misc.**Subscription**(*unsubscribe_fn: Callable[Any]*)

Bases: object

unsubscribe() → None

2.5 Release Notes

2.5.1 v0.12.0

- Change IPC backend from multiprocessing to asyncio
- Endpoint.broadcast() is now async
- Endpoint.broadcast_nowait() now exists, it schedules the message to be broadcast
- Endpoint.start_serving_nowait() no longer exists
- Endpoint.connect_to_endpoints_blocking() no longer exists
- Endpoint.stop() must be called or else some coroutines will be orphaned
- Endpoint can only be used from one event loop. It will remember the current event loop when an async method is first called, and throw an exception if another of its async methods is called from a different event loop.
- Messages will be compressed if python-snappy is installed
- Lahja previously silently dropped some exceptions, they are now propogated up

2.5.2 v0.11.2

- Properly set up logger

2.5.3 v0.11.1

- Turn exception that would be raised in a background task into a warning

2.5.4 v0.11.0

- Performance: Connect endpoints directly without central coordinator (BREAKING CHANGE)

2.5.5 v0.10.2

- Fix issue that can crash Endpoint

2.5.6 v0.10.1

- Fix issue that can crash Endpoint

2.5.7 v0.10.0

- Make *request* API accept a *BroadcastConfig*
- Add benchmarks

2.5.8 v0.9.0

- Implement "internal events"
- Rename *max* to *num_events*
- Ensure Futures are created on the correct event loop
- Ensure all consuming APIs handle cancellations well
- Don't try to propagate events after shutdown

2.6 Contributing

Thank you for your interest in contributing! We welcome all contributions no matter their size. Please read along to learn how to get started. If you get stuck, feel free to reach for help in our Gitter channel.

2.6.1 Setting the stage

Clone the Lahja repository

```
$ git clone --recursive https://github.com/ethereum/lahja.git
```

Optional: Often, the best way to guarantee a clean Python 3 environment is with virtualenv. If we don't have `virtualenv` installed already, we first need to install it via pip.

```
pip install virtualenv
```

Then, we can initialize a new virtual environment `venv`, like:

```
virtualenv -p python3 venv
```

This creates a new directory `venv` where packages are installed isolated from any other global packages.

To activate the virtual directory we have to *source* it

```
. venv/bin/activate
```

After we have activated our virtual environment, installing all dependencies that are needed to run, develop and test all code in this repository is as easy as:

```
pip install -e .[dev]
```

2.6.2 Running the tests

A great way to explore the code base is to run the tests.

We can run all tests with:

```
pytest
```

2.6.3 Code Style

When multiple people are working on the same body of code, it is important that they write code that conforms to a similar style. It often doesn't matter as much which style, but rather that they conform to one style.

To ensure your contribution conforms to the style being used in this project, we encourage you to read our style guide.

2.6.4 Type Hints

The code bases is transitioning to use type hints. Type hints make it easy to prevent certain types of bugs, enable richer tooling and enhance the documentation, making the code easier to follow.

All new code is required to land with type hints with the exception of test code that is not expected to use type hints.

All parameters as well as the return type of defs are expected to be typed with the exception of `self` and `cls` as seen in the following example.

```
def __init__(self, wrapped_db: BaseDB) -> None:
    self.wrapped_db = wrapped_db
    self.reset()
```

2.6.5 Documentation

Public APIs are expected to be annotated with docstrings as seen in the following example.

```
def add_transaction(self,
                    transaction: BaseTransaction,
                    computation: BaseComputation,
                    block: BaseBlock) -> Tuple[Block, Dict[bytes, bytes]]:
        """
        Add a transaction to the given block and
        return `trie_data` to store the transaction data in chaindb in VM layer.

        Update the bloom_filter, transaction trie and receipt trie roots, bloom_
↪filter,
        bloom, and used_gas of the block.

        :param transaction: the executed transaction
        :param computation: the Computation object with executed result
        :param block: the Block which the transaction is added in

        :return: the block and the trie_data
        """
```

Docstrings are written in reStructuredText and allow certain type of directives.

Notice that `:param:` and `:return:` directives are being used to describe parameters and return value. Usage of `:type:` and `:rtype:` directives on the other hand is discouraged as sphinx directly reads and displays the types from the source code type definitions making any further use of `:type:` and `:rtype:` obsolete and unnecessarily verbose.

Use imperative, present tense to describe APIs: "return" not "returns"

One way to test if you have it right is to complete the following sentence.

If you call this API it will: ____________________________

2.6.6 Pull Requests

It's a good idea to make pull requests early on. A pull request represents the start of a discussion, and doesn't necessarily need to be the final, finished submission.

GitHub's documentation for working on pull requests is available here.

Once you've made a pull request take a look at the Circle CI build status in the GitHub interface and make sure all tests are passing. In general pull requests that do not pass the CI build yet won't get reviewed unless explicitly requested.

2.6.7 Releasing

Pandoc is required for transforming the markdown README to the proper format to render correctly on pypi.

For Debian-like systems:

```
apt install pandoc
```

Or on OSX:

```
brew install pandoc
```

To release a new version:

```
bumpversion $$VERSION_PART_TO_BUMP$$
git push && git push --tags
make release
```

How to bumpversion

The version format for this repo is `{major}.{minor}.{patch}` for stable, and `{major}.{minor}.{patch}-{stage}.{devnum}` for unstable (`stage` can be alpha or beta).

To issue the next version in line, use bumpversion and specify which part to bump, like `bumpversion minor` or `bumpversion devnum`.

If you are in a beta version, `bumpversion stage` will switch to a stable.

To issue an unstable version when the current version is stable, specify the new version explicitly, like `bumpversion --new-version 4.0.0-alpha.1 devnum`

2.7 Code of Conduct

2.7.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

2.7.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

2.7.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

2.7.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

2.7.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at piper@pipermerriam.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

2.7.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

Python Module Index

Index