daffi

version 1.4.1

None

None

Table of contents

1. Daffi

Test and validate passing | Publish documentation passing | Coverage 76% | License MIT | code style black

downloads/month 530

Daffi facilitates remote computing and enables remote procedure calls between multiple endpoints. It supports many-to-many relationships between endpoints, allowing for seamless communication between distributed systems. The library abstracts the complexities of remote computing and provides a user-friendly interface for initiating and managing remote procedure calls. It also offers various features such as fault tolerance, load balancing, streaming and security, to ensure reliable and secure communication between endpoints.

Daffi comprises three primary classes:

- *Global* - Initialization entrypoint. Once *Global* object is initialized application can respond on remote requests and trigger remote callbacks itself.
- *Callback* - Represents a collection of methods encapsulated in a class inherited from *Callback* or a standalone function decorated with the *callback* decorator. These functions/methods can be triggered from another process.
- *Fetcher* - Represents a collection of methods encapsulated in a class inherited from *Fetcher* or a standalone function decorated with the *fetcher* decorator. These functions/methods serve as triggers for the corresponding callbacks defined in another process.

1.1 Basic example

You need to create two files `shopping_service.py` and `shopper.py`

`shopping_service.py` - represents a set of remote callbacks that can be triggered by a client application.

`shopper.py` - represents shopping_service client (fetcher)

class based approach **decorator based approach**

`shopping_service.py` :

```python
import logging
from daffi import Global
from daffi.registry import Callback

logging.basicConfig(level=logging.INFO)


class ShoppingService(Callback):
    auto_init = True # class is automatically initialized, eliminating the need to manually create an object.

    def __post_init__(self):
        self.shopping_list = []

    def get_items(self):
        """Return all items that are currently present in shopping list"""
        return self.shopping_list

    def add_item(self, item):
        """Add new item to shopping list"""
        self.shopping_list.append(item)

    def clear_items(self):
        """Clear shopping list"""
        self.shopping_list.clear()


if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888).join()
```

(This script is complete, it should run "as is")

`shopper.py` :

```python
import logging
from daffi import Global
from daffi.decorators import alias
from daffi.registry import Fetcher

logging.basicConfig(level=logging.INFO)


class Shopper(Fetcher):
    """
    Note: Functions without a body are treated as proxies for remote callbacks.
    All arguments provided to this function will be sent to the remote service as-is.
    """

    def get_items(self):
        """Return all items that are currently present in shopping list."""
        pass

    def add_item(self, item):
        """Add new item to shopping list."""
        pass

    def clear_items(self):
        """Clear shopping list"""
        pass

    @alias("add_item")
    def add_many_items(self, *items):
        """
        Alias for `add_item` callback.
        This function shows streaming capabilities for transferring data from one service to another.
        """
        for item in items:
            yield item


if __name__ == '__main__':
    g = Global(host="localhost", port=8888)

    shopper = Shopper()
    items = shopper.get_items()
    print(items)

    shopper.add_item("orange")
    items = shopper.get_items()
    print(items)

    shopper.add_many_items("bread", "cheese")
    items = shopper.get_items()
    print(items)

    shopper.clear_items()
    items = shopper.get_items()
    print(items)

    g.stop()
```

(This script is complete, it should run "as is")

To check the full example, you need to execute two scripts in separate terminals

Note

To use UNIX socket instead of TCP for communication, you should remove the `host` and `port` parameters from the initialization of the Global object, and optionally include the `unix_sock_path` parameter.

2. Usage

2.1 basic example

Example provides a basic and fundamental understanding of client-server communication

class based approach **decorator based approach**

`calculator_service.py` content:

```
import logging
from daffi import Global
from daffi.registry import Callback

logging.basicConfig(level=logging.INFO)


class CalculatorService(Callback):
    auto_init = True

    def calculate_sum(self, *numbers):
        return sum(numbers)


if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888).join()
```

`calculator_client.py` content:

```
import logging
from daffi import Global
from daffi.registry import Fetcher

logging.basicConfig(level=logging.INFO)


class CalculatorClient(Fetcher):

    def calculate_sum(self, *numbers):
        """
        Note: functions without a body are treated as proxies for remote callbacks.
        All arguments provided to this function will be sent to the remote service as-is.
        """
        pass


if __name__ == '__main__':
    g = Global(host="localhost", port=8888)

    calc_client = CalculatorClient()
    result = calc_client.calculate_sum(1, 2)
    print(result)

    result = calc_client.calculate_sum(10, 20, 30)
    print(result)

    g.stop()
```

Execute in two separate terminals:

```
python3 calculator_service.py
python3 calculator_client.py
```

`calculator_service.py` content:

```
import logging
from daffi import Global
from daffi.decorators import callback

logging.basicConfig(level=logging.INFO)


@callback
def calculate_sum(*numbers):
    return sum(numbers)


if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888).join()
```

`calculator_client.py` content:

```
import logging
from daffi import Global
from daffi.decorators import fetcher

logging.basicConfig(level=logging.INFO)


@fetcher
def calculate_sum(*numbers):
    """
    Note: functions without a body are treated as proxies for remote callbacks.
    All arguments provided to this function will be sent to the remote service as-is.
    """
    pass


if __name__ == '__main__':
    g = Global(host="localhost", port=8888)
```

Note

To use UNIX socket instead of TCP for communication, you should remove the `host` and `port` parameters from the initialization of the Global object, and optionally include the `unix_sock_path` parameter.

2.2 fetchers with function body

Fetcher functions exhibit different behavior based on whether they have a body or not. If a fetcher function does not have a body, it acts as a proxy, transmitting all provided arguments directly to the remote callback without any modification. On the other hand, if a fetcher function has a body, only the arguments returned from the function will be transmitted to the remote callback. This allows for additional intermediate logic to be executed before sending the arguments to the remote location.

Arguments can be returned as tuple or using special `Args` class.

class based approach **decorator based approach**

`calculator_service.py` content:

```
import logging
from daffi import Global
from daffi.registry import Callback

logging.basicConfig(level=logging.INFO)


class CalculatorService(Callback):
    auto_init = True

    def calculate_sum(self, num1, num2):
        return num1 + num2

    def calculate_difference(self, num1, num2):
        return num1 - num2


if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888).join()
```

`calculator_client.py` content:

```
import logging
from daffi import Global
from daffi.registry import Fetcher, Args

logging.basicConfig(level=logging.INFO)


class CalculatorClient(Fetcher):

    def __post_init__(self):
        self.multiplier = 2

    def calculate_sum(self, num1, num2):
        """Return arguments as tuple"""
        num1 *= self.multiplier
        num2 *= self.multiplier
        return num1, num2


    def calculate_difference(self, num1, num2):
        """Return arguments as `Args` class"""
        num1 *= self.multiplier
        num2 *= self.multiplier
        return Args(num1=num1, num2=num2)


if __name__ == '__main__':
    g = Global(host="localhost", port=8888)

    calc_client = CalculatorClient()
    result = calc_client.calculate_sum(1, 2)
    print(result)

    result = calc_client.calculate_difference(50, 20)
    print(result)

    g.stop()
```

Execute in two separate terminals:

```
python3 calculator_service.py
python3 calculator_client.py
```

`calculator_service.py` content:

```
import logging
from daffi import Global
from daffi.decorators import callback

logging.basicConfig(level=logging.INFO)


@callback
def calculate_sum(num1, num2):
    return num1 + num2


@callback
def calculate_difference(num1, num2):
    return num1 - num2


if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888).join()
```

`calculator_client.py` content:

```
import logging
from daffi import Global
```

Note

To use UNIX socket instead of TCP for communication, you should remove the `host` and `port` parameters from the initialization of the Global object, and optionally include the `unix_sock_path` parameter.

2.3 additional arguments for callback instantiation

The Callback class allows users to include extra initialization arguments. For this reason you need to set `auto_init` flag to False (or omit it as it is default behavior).

In this scenario, it is the user's responsibility to explicitly create an instance of the class.

`calculator_service.py` content:

```
import logging
from daffi import Global
from daffi.registry import Callback

logging.basicConfig(level=logging.INFO)


class CalculatorService(Callback):
    auto_init = False

    def __init__(self, multiplier):
        super().__init__()
        self.multiplier = multiplier

    def calculate_sum(self, num1, num2):
        num1 *= self.multiplier
        num2 *= self.multiplier
        return num1 + num2


if __name__ == '__main__':
    calc_service = CalculatorService(multiplier=3)
    Global(init_controller=True, host="localhost", port=8888).join()
```

`calculator_client.py` content:

```
import logging
from daffi import Global
from daffi.registry import Fetcher

logging.basicConfig(level=logging.INFO)


class CalculatorClient(Fetcher):

    def calculate_sum(self, num1, num2):
        pass

if __name__ == '__main__':
    g = Global(host="localhost", port=8888)

    calc_client = CalculatorClient()
    result = calc_client.calculate_sum(1, 2)
    print(result)

    g.stop()
```

Execute in two separate terminals:

```
python3 calculator_service.py
python3 calculator_client.py
```

Note

To use UNIX socket instead of TCP for communication, you should remove the `host` and `port` parameters from the initialization of the Global object, and optionally include the `unix_sock_path` parameter.

2.4 private methods

Both callbacks and fetchers can contain private methods that do not behave as remote callbacks or serve as pointers on remote callbacks (in the case of fetchers). To create such methods, users have two options:

1. They can add a leading underscore to the method name.
2. They can mark the method as local using the `local` decorator.

`calculator_service.py` content:

```
import logging
from daffi import Global
from daffi.registry import Callback
from daffi.decorators import local

logging.basicConfig(level=logging.INFO)


class CalculatorService(Callback):
    auto_init = True

    def calculate_sum(self, num1, num2):
        return self.get_sum(num1, num2)

    @local
    def get_sum(self, num1, num2):
        return num1 + num2


if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888).join()
```

`calculator_client.py` content:

```
import logging
from daffi import Global
from daffi.registry import Fetcher, Args
from daffi.decorators import local

logging.basicConfig(level=logging.INFO)


class CalculatorClient(Fetcher):

    def calculate_sum(self, num1, num2):
        return self.create_args(num1, num2)

    @local
    def create_args(self, num1, num2):
        return Args(num1, num2)


if __name__ == '__main__':
    g = Global(host="localhost", port=8888)

    calc_client = CalculatorClient()
    result = calc_client.calculate_sum(1, 2)
    print(result)

    g.stop()
```

Execute in two separate terminals:

```
python3 calculator_service.py
python3 calculator_client.py
```

Note

To use UNIX socket instead of TCP for communication, you should remove the `host` and `port` parameters from the initialization of the Global object, and optionally include the `unix_sock_path` parameter.

2.5 aliased methods

Both callbacks and fetchers can contain aliased methods, which means that the methods appear under different names on the remote side. This is particularly useful for fetchers, as it allows multiple fetchers with different names and internal logic to point to a single callback.

class based approach **decorator based approach**

`calculator_service.py` content:

```
import logging
from daffi import Global
from daffi.registry import Callback

logging.basicConfig(level=logging.INFO)


class CalculatorService(Callback):
    auto_init = True

    def calculate_sum(self, num1, num2):
        return num1 + num2


if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888).join()
```

`calculator_client.py` content:

```
import logging
from daffi import Global
from daffi.registry import Fetcher, Args
from daffi.decorators import alias

logging.basicConfig(level=logging.INFO)


class CalculatorClient(Fetcher):

    def __post_init__(self):
        self.multiplier = 2

    def calculate_sum(self, num1, num2):
        """Default proxy behavior"""
        pass

    @alias("calculate_sum")
    def calculate_sum_with_multiplier(self, num1, num2):
        """Alias to the same `calculate_sum` callback but with different internal logic"""
        num1 *= self.multiplier
        num2 *= self.multiplier
        return Args(num1, num2)


if __name__ == '__main__':
    g = Global(host="localhost", port=8888)

    calc_client = CalculatorClient()
    result = calc_client.calculate_sum(1, 2)
    print(result)

    result = calc_client.calculate_sum_with_multiplier(1, 2)
    print(result)

    g.stop()
```

Execute in two separate terminals:

```
python3 calculator_service.py
python3 calculator_client.py
```

`calculator_service.py` content:

```
import logging
from daffi import Global
from daffi.decorators import callback

logging.basicConfig(level=logging.INFO)


@callback
def calculate_sum(num1, num2):
    return num1 + num2


if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888).join()
```

`calculator_client.py` content:

```
import logging
from daffi import Global
from daffi.decorators import fetcher, alias
from daffi.registry import Args

logging.basicConfig(level=logging.INFO)

multiplier = 2


@fetcher
```

Note

To use UNIX socket instead of TCP for communication, you should remove the `host` and `port` parameters from the initialization of the Global object, and optionally include the `unix_sock_path` parameter.

2.6 named processes

A process can be given a meaningful name by the user when initializing the Global object with the `process_name` argument. By default, the name is generated automatically. Naming processes can be useful in situations where one process needs to wait for another to start.

`calculator_service.py` content:

```
import logging
from daffi import Global

logging.basicConfig(level=logging.INFO)

if __name__ == '__main__':
    g = Global(init_controller=True, host="localhost", port=8888, process_name="calculator service")
    g.wait_process(process_name="calculator client")

    print("Calculator client has been started...")

    g.join()
```

`calculator_client.py` content:

```
import time
import logging
from daffi import Global

logging.basicConfig(level=logging.INFO)

if __name__ == '__main__':
    g = Global(host="localhost", port=8888, process_name="calculator client")

    time.sleep(5)

    g.stop()
```

Execute in two separate terminals:

```
python3 calculator_service.py
python3 calculator_client.py
```

Note

To use UNIX socket instead of TCP for communication, you should remove the `host` and `port` parameters from the initialization of the Global object, and optionally include the `unix_sock_path` parameter.

2.7 stream from fetcher to callback

In the world of Daffi, yield statements hold a unique significance as they pertain to stream processing. Streams can be classified into two types: those that go from a fetcher to a callback and those that go from a callback to a fetcher. Similar to return statements, yield statements can be followed by a tuple or a special "Args" class to send multiple arguments to a remote location.

Streams can also be utilized like events to wait for specific conditions on a remote.

class based approach **decorator based approach**

`stream_service.py` content:

```
import logging
from daffi import Global
from daffi.registry import Callback

logging.basicConfig(level=logging.INFO)


class StreamerService(Callback):
    auto_init = True

    def __post_init__(self):
        self.items = []

    def stream_to_service(self, item, process_name):
        self.items.append(item)
        print(f"Received item: {item} from process: {process_name}")

    def get_items(self):
        return self.items


if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888).join()
```

`stream_client.py` content:

```
import logging
from daffi import Global
from daffi.registry import Fetcher, Args

logging.basicConfig(level=logging.INFO)

PROCESS_NAME = "streamer client"


class StreamerClient(Fetcher):

    def __post_init__(self):
        self.items = range(1000)

    def stream_to_service(self):
        """Process stream"""
        for item in self.items:
            yield Args(item=item, process_name=PROCESS_NAME)

    def get_items(self):
        """Get all items from service"""
        pass


if __name__ == '__main__':
    g = Global(host="localhost", port=8888, process_name=PROCESS_NAME)

    stream_client = StreamerClient()
    stream_client.stream_to_service()

    # get all items from service after stream processing
    items = stream_client.get_items()
    print(items)

    g.stop()
```

Execute in two separate terminals:

```
python3 stream_service.py
python3 stream_client.py
```

`stream_service.py` content:

```
import logging
from daffi import Global
from daffi.decorators import callback

logging.basicConfig(level=logging.INFO)

items = []


@callback
def stream_to_service(item, process_name):
    items.append(item)
    print(f"Received item: {item} from process: {process_name}")


@callback
def get_items():
    return items

if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888).join()
```

Note

To use UNIX socket instead of TCP for communication, you should remove the `host` and `port` parameters from the initialization of the Global object, and optionally include the `unix_sock_path` parameter.

2.8 stream from callback to fetcher

Just like how streams can be initiated from a fetcher to a callback, callbacks can also initialize streams using yield statements. In this scenario, the fetcher receives a generator as the result.

Streams can also be utilized like events to wait for specific conditions on a remote.

class based approach **decorator based approach**

`stream_service.py` content:

```
import logging
from daffi import Global
from daffi.registry import Callback

logging.basicConfig(level=logging.INFO)


class StreamerService(Callback):
    auto_init = True

    def generate_stream(self, end):
        for i in range(end):
            yield i


if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888).join()
```

`stream_client.py` content:

```
import logging
from daffi import Global
from daffi.registry import Fetcher

logging.basicConfig(level=logging.INFO)


class StreamerClient(Fetcher):

    def generate_stream(self, end):
        """Generate stream by callback"""
        pass


if __name__ == '__main__':
    g = Global(host="localhost", port=8888)

    stream_client = StreamerClient()
    result = stream_client.generate_stream(end=1000)

    for item in result:
        print(item)

    g.stop()
```

Execute in two separate terminals:

```
python3 stream_service.py
python3 stream_client.py
```

`stream_service.py` content:

```
import logging
from daffi import Global
from daffi.decorators import callback

logging.basicConfig(level=logging.INFO)


@callback
def generate_stream(end):
    for item in range(end):
        yield item


if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888).join()
```

`stream_client.py` content:

```
import logging
from daffi import Global
from daffi.decorators import fetcher

logging.basicConfig(level=logging.INFO)


@fetcher
def generate_stream(end):
    pass


if __name__ == '__main__':
    g = Global(host="localhost", port=8888)

    result = generate_stream(1000)
    for item in result:
        print(item)

    g.stop()
```

Note

To use UNIX socket instead of TCP for communication, you should remove the `host` and `port` parameters from the initialization of the Global object, and optionally include the `unix_sock_path` parameter.

2.9 function transfer

Even if a particular callback is not present, it is still feasible to transfer and execute a function on the remote side. In order to utilize this feature, processes need to have static names.

`executor.py` content:

```python
import logging
from daffi import Global

logging.basicConfig(level=logging.INFO)

PROCESS_NAME = "remote executor"


if __name__ == "__main__":
    Global(process_name=PROCESS_NAME, init_controller=True, host="localhost", port=8888).join()
```

`client.py` content:

```python
import logging
from daffi import Global

logging.basicConfig(level=logging.INFO)


async def func_to_transfer():
    """
    Return pid id of remote process.
    If it's uncertain whether an import exists on the remote side,
    it's preferable to explicitly import the used libraries.
    """
    import os
    return os.getpid()


if __name__ == "__main__":
    remote_proc = "remote executor"
    g = Global(host="localhost", port=8888)

    remote_pid = g.transfer_and_call(remote_process=remote_proc, func=func_to_transfer)
    print(f"Pid of remote process: {remote_pid}")
```

Execute in two separate terminals:

```
python3 executor.py
python3 client.py
```

Note

To use UNIX socket instead of TCP for communication, you should remove the `host` and `port` parameters from the initialization of the Global object, and optionally include the `unix_sock_path` parameter.

2.10 bidirectional communication

Daffi is intended for two-way communication, with each process capable of having multiple fetchers and callbacks. There are no explicitly defined servers or clients. By taking advantage of this flexibility, users can establish intricate relationships between numerous microservices within a system.

`shopping_service.py` content:

```python
import logging
from daffi import Global
from daffi.registry import Callback, Fetcher

logging.basicConfig(level=logging.INFO)

PROCESS_NAME = "shopping service"


class ShoppingService(Callback):
    auto_init = True

    def __post_init__(self):
        self.shop_items = ["orange", "bread", "cheese"]

    def get_shop_items(self):
        return self.shop_items


class CalculatorClient(Fetcher):

    def calculate_sum(self, *numbers):
        """Proxy fetcher"""
        pass


if __name__ == '__main__':
    g = Global(init_controller=True, host="localhost", port=8888, process_name=PROCESS_NAME)
    # Wait counterpart process
    g.wait_process("calculator service")

    calc_client = CalculatorClient()
    _sum = calc_client.calculate_sum(1, 2, 3)
    print(f"Calculated sum: {_sum}")

    g.join()
```

`calculator_service.py` content:

```python
import logging
from daffi import Global
from daffi.registry import Callback, Fetcher

logging.basicConfig(level=logging.INFO)

PROCESS_NAME = "calculator service"


class CalculatorService(Callback):
    auto_init = True

    def calculate_sum(self, *numbers):
        return sum(numbers)


class ShoppingClient(Fetcher):

    def get_shop_items(self):
        """Proxy fetcher"""
        pass


if __name__ == '__main__':
    g = Global(host="localhost", port=8888, process_name=PROCESS_NAME)
    # Wait counterpart process
    g.wait_process("shopping service")

    shopping_client = ShoppingClient()
    items = shopping_client.get_shop_items()
    print(f"Received shopping items: {items}")

    g.join()
```

Execute in two separate terminals:

```
python3 shopping_service.py
python3 calculator_service.py
```

Note

To use UNIX socket instead of TCP for communication, you should remove the `host` and `port` parameters from the initialization of the Global object, and optionally include the `unix_sock_path` parameter.

2.11 SSL certificates

Communication between daffi applications can be protected with SSL certificate and key.

For this example consider to use following script in order to generate self signed certificates for localhost IPV6

```
#!/bin/bash
# Generate self-signed certificates

set -e

IP="::"

openssl req -x509 -newkey rsa:4096 -sha256 -days 3650 -nodes \
  -keyout key.pem -out cert.pem -subj "/CN=example.com" \
  -addext "subjectAltName=DNS:example.com,DNS:www.example.net,IP:${IP}"
```

As the result you should see `cert.pem` and `key.pem` files created in directory where script was executed.

class based approach **decorator based approach**

`calculator_service.py` content:

```python
import logging
from daffi import Global
from daffi.registry import Callback

logging.basicConfig(level=logging.INFO)


class CalculatorService(Callback):
    auto_init = True

    def calculate_sum(self, *numbers):
        return sum(numbers)


if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888, ssl_certificate="cert.pem", ssl_key="key.pem").join()
```

`calculator_client.py` content:

```python
import logging
from daffi import Global
from daffi.registry import Fetcher

logging.basicConfig(level=logging.INFO)


class CalculatorClient(Fetcher):

    def calculate_sum(self, *numbers):
        """
        Note: functions without a body are treated as proxies for remote callbacks.
        All arguments provided to this function will be sent to the remote service as-is.
        """
        pass


if __name__ == '__main__':
    g = Global(host="localhost", port=8888, ssl_certificate="cert.pem", ssl_key="key.pem")

    calc_client = CalculatorClient()
    result = calc_client.calculate_sum(1, 2)
    print(result)

    result = calc_client.calculate_sum(10, 20, 30)
    print(result)

    g.stop()
```

Execute in two separate terminals:

```
python3 calculator_service.py
python3 calculator_client.py
```

`calculator_service.py` content:

```python
import logging
from daffi import Global
from daffi.decorators import callback

logging.basicConfig(level=logging.INFO)


@callback
def calculate_sum(*numbers):
    return sum(numbers)


if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888, ssl_certificate="cert.pem", ssl_key="key.pem").join()
```

`calculator_client.py` content:

```python
import logging
from daffi import Global
from daffi.decorators import fetcher

logging.basicConfig(level=logging.INFO)


@fetcher
def calculate_sum(*numbers):
    """
    Note: functions without a body are treated as proxies for remote callbacks.
    All arguments provided to this function will be sent to the remote service as-is.
    """
    pass


if __name__ == '__main__':
    g = Global(host="localhost", port=8888, ssl_certificate="cert.pem", ssl_key="key.pem")
```

Note

To use UNIX socket instead of TCP for communication, you should remove the `host` and `port` parameters from the initialization of the Global object, and optionally include the `unix_sock_path` parameter.

2.12 lifecycle events

Global object takes 3 lifecycle callbacks

- `on_init` executed once Node, Controller or Controller and Node are initialized. It is one time event
- `on_node_connect` - callback is triggered when the Node establishes a connection with the Controller and can occur multiple times, including instances where re-connections are involved
- `on_node_disconnect` - callback is triggered when the Node is disconnected from Controller and can occur multiple times, including instances where re-connections are involved

```python
import logging
from daffi import Global

logging.basicConfig(level=logging.INFO)


def on_init(g: Global, process_name: str):
    print("On init")


def on_node_connect(g: Global, process_name: str):
    print("Node connected")


def on_node_disconnect(g: Global, process_name: str):
    print("Node disconnected")


if __name__ == "__main__":
    Global(
        init_controller=True,
        host="localhost",
        port=8888,
        on_init=on_init,
        on_node_connect=on_node_connect,
        on_node_disconnect=on_node_disconnect,
    ).join()
```

2.13 asynchronous execution

Daffi utilizes classes known as execution modifiers to transmit requests to a remote process. These modifiers define the manner in which requests should be executed and how the system should await the resulting computation. The default modifier, which we have implicitly used in previous examples, is `FG`, which stands for "foreground." This modifier blocks the main process until the computed result is returned.

In the following example, we will use the `BG` execution modifier, which is short for "background." This modifier returns an instance of `AsyncResult` rather than the result itself, making it appropriate for long-running tasks or tasks in which the result is not critical.

class based approach **decorator based approach**

`calculator_service.py` content:

```
import logging
from daffi import Global
from daffi.registry import Callback

logging.basicConfig(level=logging.INFO)


class CalculatorService(Callback):
    auto_init = True

    def calculate_fibonacci(self, n):
        # Check if n is 0
        # then it will return 0
        if n == 0:
            return 0

        # Check if n is 1,2
        # it will return 1
        elif n == 1 or n == 2:
            return 1

        else:
            return self.calculate_fibonacci(n - 1) + self.calculate_fibonacci(n - 2)


if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888).join()
```

`calculator_client.py` content:

```
import time
import logging
from daffi import Global, BG, FG
from daffi.registry import Fetcher

logging.basicConfig(level=logging.INFO)


class CalculatorClient(Fetcher):

    # Execution modifier are shared across all fetcher's methods
    exec_modifier = BG(timeout=30)

    def calculate_fibonacci(self, n):
        """Proxy fetcher"""
        pass


if __name__ == '__main__':
    g = Global(host="localhost", port=8888)

    calc_client = CalculatorClient()
    feature = calc_client.calculate_fibonacci(20)

    print("Waiting")
    time.sleep(10)

    result = feature.get()
    print(f"Fibonacci result: {result}")

    # Execution modifier can be specified at execution time
    result = calc_client.calculate_fibonacci.call(exec_modifier=FG, n=15)
    print(f"Fibonacci result: {result}")

    g.stop()
```

Execute in two separate terminals:

```
python3 calculator_service.py
python3 calculator_client.py
```

`calculator_service.py` content:

```
import logging
from daffi import Global
from daffi.decorators import callback

logging.basicConfig(level=logging.INFO)


@callback
def calculate_fibonacci(n):
    # Check if n is 0
    # then it will return 0
    if n == 0:
        return 0

    # Check if n is 1,2
    # it will return 1
    elif n == 1 or n == 2:
        return 1

    else:
```

Note

To use UNIX socket instead of TCP for communication, you should remove the `host` and `port` parameters from the initialization of the Global object, and optionally include the `unix_sock_path` parameter.

2.14 broadcasting

You can register callbacks with the same name on multiple processes, allowing for simultaneous execution. In addition to one-to-one communication, there is a specific execution modifier called `BROADCAST` that can be utilized in fetchers. The `BROADCAST` modifier ensures that the result is only returned after all callbacks have completed their work.

The resulting output takes on a dictionary structure, with keys representing the process names and corresponding values representing the computed results.

```
{
    "process-1":  "<result-1>",
    "process-2":  "<result-2>",
    "process-N":  "<result-N>"
}
```

Warning

Make sure you initialized only one process with `init_controller=True` argument

class based approach **decorator based approach**

`burger_menu_service.py` content:

```
import logging
from daffi import Global
from daffi.registry import Callback

logging.basicConfig(level=logging.INFO)


class BurgerMenu(Callback):
    auto_init = True

    def get_menu(self):
        return ["The IceBurg", "The Grill Thrill", "Burger Mania", "Chicha Burger"]


if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888, process_name="burger menu").join()
```

`hotdog_menu_service.py` content:

```
import logging
from daffi import Global
from daffi.registry import Callback

logging.basicConfig(level=logging.INFO)


class HotDogMenu(Callback):

    def get_menu(self):
        return ["Wiener", "Weenie", "Coney", "Red Hot"]


if __name__ == '__main__':
    Global(host="localhost", port=8888, process_name="hotdog menu").join()
```

`menu_client.py` content:

```
import logging
from daffi import Global, BROADCAST
from daffi.registry import Fetcher

logging.basicConfig(level=logging.INFO)


class MenuFetcher(Fetcher):
    exec_modifier = BROADCAST

    def get_menu(self):
        pass


if __name__ == '__main__':
    menu_fetcher = MenuFetcher()
    g = Global(host="localhost", port=8888)

    # Make sure all processes started
    for proc in ("burger menu", "hotdog menu"):
        g.wait_process(proc)

    menus = menu_fetcher.get_menu()
    print(menus)
    # {'burger menu': ['The IceBurg', 'The Grill Thrill', 'Burger Mania', 'Chicha Burger'], 'hotdog menu': ['Wiener', 'Weenie', 'Coney', 'Red Hot']}

    g.stop()
```

Execute in three separate terminals:

```
python3 burger_menu_service.py
python3 hotdog_menu_service.py
python3 menu_client.py
```

`burger_menu_service.py` content:

```
import logging
from daffi import Global
from daffi.decorators import callback

logging.basicConfig(level=logging.INFO)


@callback
def get_menu():
    return ["The IceBurg", "The Grill Thrill", "Burger Mania", "Chicha Burger"]


if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888, process_name="burger menu").join()
```

`hotdog_menu_service.py` content:

Note

To use UNIX socket instead of TCP for communication, you should remove the `host` and `port` parameters from the initialization of the Global object, and optionally include the `unix_sock_path` parameter.

2.15 task scheduling and batch delayed execution

Using a special class modifier called `PERIOD`, it is possible to initiate recurring callback executions or schedule a group of delayed callback executions simultaneously.

`PERIOD` take 2 optional arguments `at_time` and `interval` but only one of them is allowed to be provided.

EXAMPLE

class based approach **decorator based approach**

`scheduler.py` content:

```
import time
import logging
from daffi import Global
from daffi.registry import Callback

logging.basicConfig(level=logging.INFO)


class Scheduler(Callback):
    auto_init = True

    def long_running_task1(self):
        print("Start long running task 1")
        for i in range(1, 3):
            print(f"Processing item {i}...")
            time.sleep(1)
        print("Task 1 complete.")

    def long_running_task2(self):
        print("Start long running task 2")
        for i in range(100, 103):
            print(f"Processing item {i}...")
            time.sleep(1)
        print("Task 2 complete.")


if __name__ == '__main__':
    Global(init_controller=True, host="localhost", port=8888).join()
```

`scheduler_client.py` content:

```
import time
import logging
from datetime import datetime
from daffi import Global, PERIOD
from daffi.registry import Fetcher

logging.basicConfig(level=logging.INFO)


class SchedulerClient(Fetcher):
    # The default behavior of this fetcher is to execute each of its methods every 5 seconds.
    exec_modifier = PERIOD(interval="5s")

    def long_running_task1(self):
        pass

    def long_running_task2(self):
        pass


if __name__ == '__main__':
    g = Global(host="localhost", port=8888)
    scheduler_client = SchedulerClient()

    task_ident = scheduler_client.long_running_task1()
    time.sleep(30)

    # Stop the current recurring task.
    task_ident.cancel()

    now = datetime.utcnow().timestamp()
    at_time = [now + 10, now + 30]

    # Execute the `long_running_task2` function twice on the remote.
    # The first execution will occur 10 seconds from now, and the second will occur 30 seconds from now.
    task_ident = scheduler_client.long_running_task2.call(exec_modifier=PERIOD(at_time=at_time))

    g.stop()
```

Execute in two separate terminals:

```
python3 scheduler.py
python3 scheduler_client.py
```

`scheduler.py` content:

```
import time
import logging
from daffi import Global
from daffi.decorators import callback

logging.basicConfig(level=logging.INFO)



@callback
def long_running_task1():
    print("Start long running task 1")
    for i in range(1, 3):
        print(f"Processing item {i}...")
        time.sleep(1)
    print("Task 1 complete.")
```

`AT_TIME` ARGUMENT

You can provide either a single timestamp or a list of timestamps, all of which must be related to UTC time and in the future.

For example, if you want to execute the remote callback `some_func` three times - 2 seconds from now, 10 seconds from now, and 1 minute from now - you would specify these three timestamps:

```
import logging
from datetime import datetime
from daffi import Global, PERIOD
from daffi.decorators import fetcher

logging.basicConfig(level=logging.INFO)


g = Global(host="localhost", port=8888)


@fetcher
def some_func():
    pass


# Wait my_callback function to be available on remote
g.wait_function("some_func")

now = datetime.utcnow().timestamp()

# 2 sec, 10 sec and 60 sec later
at_time = [now + 2, now + 10, now + 60]

task = some_func.call(exec_modifier=PERIOD(at_time=at_time))
```

Execution with `PERIOD` modifier returns `ScheduledTask` instance as result of execution.

You can cancel all tasks triggered from this execution:

```
...
time.sleep(3)
task.cancel()
```

As we slept 3 seconds before canceling only two executions that remains in `at_time` bunch will be canceled as first of them was already triggered earlier.

`INTERVAL` ARGUMENT

Using `interval` has the same execution signature and also returns instance of `ScheduledTask` as result so you can cancel reccuring task any time you want

`PERIOD` with `interval` argument has the same execution signature and also returns an instance of `ScheduledTask`, allowing you to cancel the recurring task at any time.

```
import time
import logging
from datetime import datetime
from daffi import Global, PERIOD
from daffi.decorators import fetcher

logging.basicConfig(level=logging.INFO)


@fetcher(PERIOD(interval=5))
def some_func():
    pass


g = Global(host="localhost", port=8888)

# Wait my_callback function to be available on remote
g.wait_function("some_func")

now = datetime.utcnow().timestamp()

# Execute remote callback `some_func` each 5 seconds
task = some_func()
time.sleep(60)
task.cancel()
```

On example above we started recurring execution each 5 second and canceled it after sleep

Note

interval also takes special string formatted expressions as values for instance next statement is also valid:

PERIOD(interval="5s")

Another examples:

PERIOD(interval="1m24s") # == 84 seconds

PERIOD(interval="1.2 minutes") # == 72 seconds

3. Node and Controller

3.1 Controller

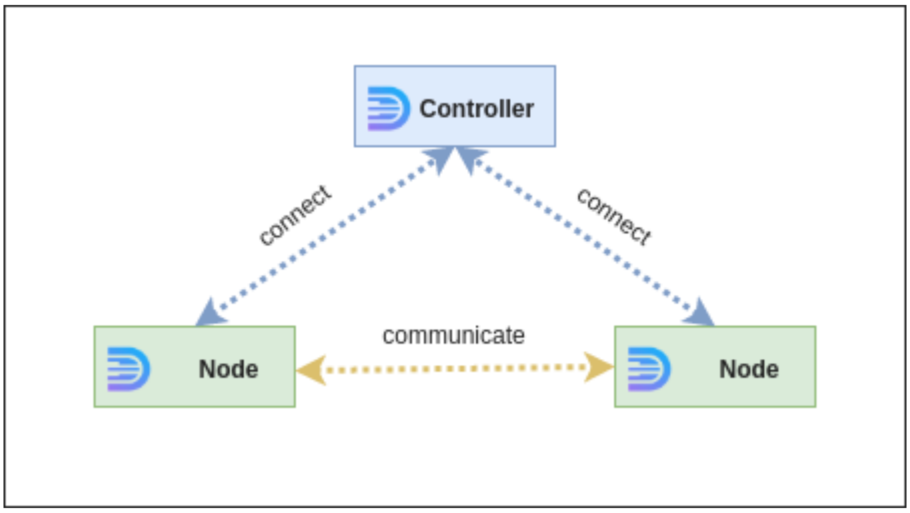
In Daffi's parlance, the server is referred to as the `Controller.` This Controller operates as an intermediary and is not able to independently invoke remote callbacks. It can be launched within any process in which remote callbacks are registered, or it can function as a standalone application. The choice between these options depends on the user's specific needs. Typically, an application infrastructure will feature only one Controller, although multiple Controllers can be used to create isolated systems if desired.

3.2 Node

A `Node` in Daffi is a client that runs either alongside the Controller or as a separate process. When a Node is running, its associated application can register callbacks and invoke the callbacks of other Nodes. The Node is responsible for managing all aspects of serialization/deserialization, remote callback execution, and related processes.

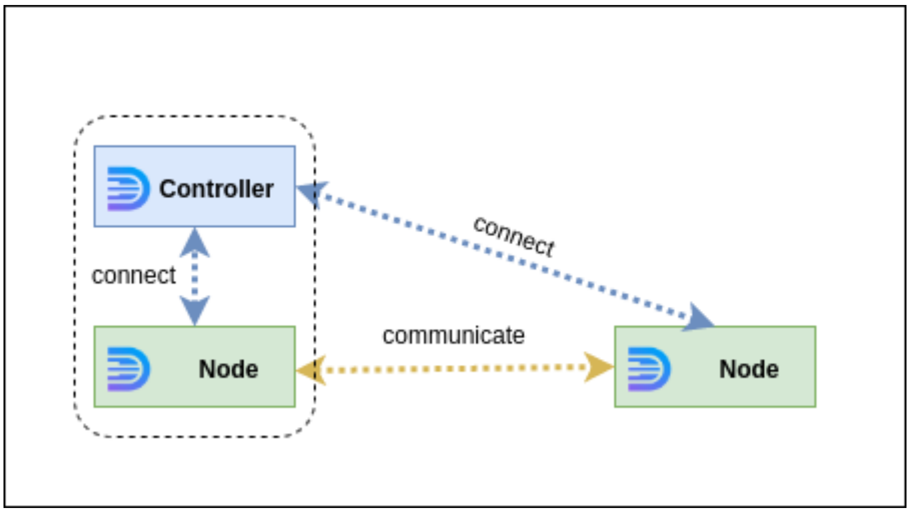
3.2.1 Typical architecture

A typical application architecture consists of one controller and two or more nodes. The controller can work as a standard application or share the process with a node:



Stand alone `Controller`.

This architecture is suitable where there are more than 2 nodes and complex inter-node communication is expected.



`Controller` shares process with one of nodes.

This architecture is suitable for simple node-to-node requests or streams. But also can be considered as solution when one of the processes is the leader with the ability to give commands to all other nodes

Warning

If you want to initialize controller with the ability to make remote requests, you need to consider latter solution (`Controller shares process with one of nodes`)

4. Global object

Global is the main initialization daffi entrypoint.

Global object initialization

You can initialize Global object using the following syntax:

```
from daffi import Global
g = Global(process_name='my awersome process', init_controller=True, host='localhost', port=8888)
```

where:

The `process_name` argument serves as an optional identifier for the process, allowing other processes to identify the initialized node by its given process_name.

If the process_name argument is not provided, a randomly generated name will be used. However, assigning meaningful names to nodes can be beneficial in certain scenarios. For example, one process may need to wait for another process by its specified name.

```
g.wait_process('node name here')
```

`init_controller=True` Means we want to start `Controller` in this process.

`init_node` argument is True by default so if you want to start only controller in particular process you should be explicit:

```
g = Global(process_name=process_name, init_controller=True, init_node=False, host='localhost', port=8888)
```

`host` and `port` arguments give `Controller` and `Node` information how to connect and communicate.

`host` and `port` arguments are also optional. For instance you can specify only `host`. In this case `Controller` / `Node` or both will be connected to random port

You can also skip these two arguments:

```
g = Global(process_name=process_name, init_controller=True)
```

In this case `Controller` / `Node` will be connected using UNIX socket. By default UNIX socket is created within path

```
< temp directory >/daffi/.sock
```

Where `< temp directory >` is temporary directory of machine where `Controller` / `Node` is running. For instance it is going to be `/tmp/daffi/.sock` on Ubuntu.

You can also provide your own directory for UNIX socket:

```
g = Global(process_name=process_name, init_controller=True, unix_sock_path="/foo/bar/biz")
```

Execution workflow

After initialization Global object starts `Controller` / `Node` or both in separate thread.

You can join this thread to main process using `.join` of `.join_async` methods of Global

```
g = Global(process_name=process_name, init_controller=True, init_node=False, host='localhost', port=8888)
g.join()

#--- or `join_async` if your application is asynchronous
await g.join_async()
```

`Controller` / `Node` or both can be terminated by calling the `stop` method. This method is particularly useful for short-lived jobs, such as starting a Daffi process, triggering a few callbacks on other nodes, and then terminating the process.

```
g = Global(process_name=process_name, init_controller=True, init_node=False, host='localhost', port=8888)
# .....
# Execute remote callbacks ...
g.stop()
```

For this reason Global can be used as context manager. `stop` method is executed explicitly on exit from context manager scope.

```
with Global(process_name=process_name, init_controller=True, init_node=False, host='localhost', port=8888) as g:
    # .....
    # Execute remote callbacks ...
```

Warning

Dont use `.stop` method to stop daffi components in long living applications eg web servers, background workers etc. start/stop daffi components requires some initialization time and resources.

waiting for nodes or methods to be available

Sometimes nodes start at different times and because of this, some remote callbacks may not be available immediately.

Global has several methods to control waiting for callbacks availability.

The 2 examples below illustrate waiting for a remote process to be available:

```
g.wait_process('name of remote node')
```

or

```
await g.wait_process_async('name of remote node')
```

Global can also wait a specific callback to be available by its name:

```
g.wait_function('name of remote callback')
```

or

```
await g.wait_function_async('name of remote callback')
```

Waiting by callback name criteria can be useful when many nodes contain a callback with the same name and we need to wait for the presence of one of them

Transfer and execute function on remote Node

This option is suitable when you want to create remote callback dynamically and execute them on remote nodes.

Example:

process 1

```
import logging
from daffi import Global

logging.basicConfig(level=logging.INFO)


def main():
    Global(process_name="other_process", init_controller=True, host="localhost", port=8888).join()


if __name__ == "__main__":
    main()
```

(This script is complete, it should run "as is")

process 2

```
import logging
from daffi import Global

logging.basicConfig(level=logging.INFO)


async def func_to_transfer():
    """Return pid id of remote process"""
    import os

    return os.getpid()


def main():

    with Global(host="localhost", port=8888) as g:

        remote_pid = g.transfer_and_call(remote_process="other_process", func=func_to_transfer)
        print(f"Remote process pid: {remote_pid}")


if __name__ == "__main__":
    main()
```

(This script is complete, it should run "as is")

On example above we transfer function `func_to_transfer` from process2 to process1 and execute it. The result of function execution will be returned to process2.

Warning

You should make sure all imports that are using in function body are available on remote process. Otherwise you should consider import modules in function body (like `import os` in example above).

Working with scheduled tasks

Global has methods `get_scheduled_tasks` and `cancel_scheduled_task_by_uuid` to works with scheduled tasks (see scheduling tasks) on remote process.

- `get_scheduled_tasks` get all scheduled tasks on remote process by process name.The method returns all task UUIDs that are currently scheduled on the remote process.
- `cancel_scheduled_task_by_uuid` cancel one scheduled task on remote process by its UUID. All UUIDs can be obtained using `get_scheduled_tasks` method.

5. Asynchronous applications

The majority of Daffi's methods are compatible with both synchronous and asynchronous applications, but certain methods may impede the event loop. In such cases, it is recommended to use the asynchronous counterparts of these blocking methods.

For instance, the Global object's `join` method is a blocking method that cannot be used with event loop-dependent applications. However, the `join_async` method is an asynchronous, non-blocking alternative. There are two methods available for retrieving results: `get` and `get_async`, depending on the application type.

As example we can use `BG` execution modifier (run in background) which returns instance of `AsyncResult` instead of result itself:

```
from daffi import BG
from daffi.decorators import fetcher

@fetcher(BG)
def my_callback():
    pass

future = my_callback()

# get result in blocking manner
result = future.get()

# get result in async applications
result = await future.get_async()
```

Despite on remote callback type we can declare synchronous or asynchronous fetcher depends on our application.

For instance if in `process-1` we have callback with name `trigger_me`:

process-1

```
from daffi.decorators import callback

@callback
def trigger_me():
    print("Triggered")
```

In `process-2` we can declare sync or async fetcher to trigger callback:

process-2

```
from daffi import FG
from daffi.decorators import fetcher

@fetcher(FG)
def trigger_me():
    pass

def main():
    trigger_me()

# ....
```

The more appropriate syntax for asynchronous applications will be:

process-2

```
from daffi import FG
from daffi.decorators import fetcher

@fetcher(FG)
async def trigger_me():
    pass

async def main():
    await trigger_me()

# ....
```

6. Execution modifiers

Execution modifiers are set of classes that specify the manner in which the remote request should be carried out, as well as how the process executor should anticipate the resulting computation.

list of available execution modifiers:

Modifier class	**Description**	**Optional arguments**
`FG`	Stands for `foreground`. It means current process execution should be blocked until result of execution is returned	- `timeout`: The time to wait result. If exeeded `TimeoutError` will be thrown
`BG`	Stands for `background`. This modifier returns `AsyncResult` instance instead of result. Fits for long running tasks where caller execution cannot be blocked for a long time.	- `timeout`: The time to wait result. If exeeded `TimeoutError` will be thrown - `eta`: The time to sleep in the background before sending execution request - `return_result`: boolean flag that indicates if we want to receive result of remote callback execution.
`PERIOD`	Use for scheduling reccuring tasks or tasks which should be executed several times.	- `at_time`: One timestamp or list of timestamps. Timestamps should be according to utc time and it should be timestamp in the future. This argument forces remote callback to be triggered one or more times when timestamp == datetime.utcnow().timestamp - `period`: Duration in seconds to trigger remote callback on regular bases. One can provide either `at_time` argument or `period` argument in one request. Not both!
`BROADCAST`	Trigger all available callbacks on nodes by name. If `return_result` argument is set to True then aggregated result will be returned as dictionary where keys are node names and values are computed results.	- `timeout`: The time to wait result. If exeeded `TimeoutError` will be thrown - `eta`: The time to sleep in the background before sending execution request `return_result`: If provided aggregated result from all nodes where callback exist will be returned.

7. Code reference

7.1 Global

The main entry point for all remote operations, such as calling remote callbacks, waiting for remote processes, and obtaining additional information from remote sources.

Parameters:

Name	Type	Description	Default
`process_name`	`Optional[str]`	Global process name. If specified it is used as reference key to Node process. By default randomly generated hash is used as reference.	`None`
`init_controller`	`Optional[bool]`	Flag that indicates whether `Controller` should be instantiated in current process	`False`
`init_node`	`Optional[bool]`	Flag that indicates whether `Node` should be instantiated in current process	`True`
`host`	`Optional[str]`	host to connect `Controller` / `Node` via tcp. If not provided then Global consider UNIX socket connection to be used.	`None`
`port`	`Optional[int]`	Optional port to connect `Controller` / `Node` via tcp. If not provided random port will be chosen.	`None`
`unix_sock_path`	`Optional[os.PathLike]`	Folder where UNIX socket will be created. If not provided default path is < tmp directory >/dafi/ where `<tmp directory >` is default temporary directory on system.	`None`
`on_init`	`Optional[Callable[[Global, str], Any]]`	Function that will be executed once when Global object is initialized. `on_init` takes Global object as first argument	`None`
`on_node_connect`	`Optional[Callable[[Global, str], Any]]`	Function that will be executed each time when connection to Controller is established (IOW it works only for Nodes). `on_connect` takes Global object as first argument	`None`
`on_node_disconnect`	`Optional[Callable[[Global, str], Any]]`	Function that will be executed each time when connection to Controller is lost	`None`

7.1.1 `call: LazyRemoteCall` property

Returns instance of `LazyRemoteCall` that is used to trigger remote callback.

7.1.2 `is_controller: bool` property

Return True if controller is running in current process

7.1.3 `registered_callbacks: Dict[str, List[str]]` property

Return list of all registered callbacks along with process names where these callbacks are registered.

7.1.4 `cancel_scheduled_task_by_uuid(remote_process: str, uuid: int) -> NoReturn`

Cancel scheduled task by its uuid on remote process. Find out task uuid you can using `get_scheduled_tasks` method.

7.1.5 `get_scheduled_tasks(remote_process: str)`

Get all scheduled tasks that are running at this moment on remote process

Parameters:

Name	Type	Description	Default
`remote_process`	`str`	Name of `Node`	*required*

7.1.6 `join() -> NoReturn`

Join global to main thread. Don't use this method if you're running asynchronous application as it blocks event loop.

7.1.7 `join_async()` `async`

Async version of .join() method. Use this method if your application is asynchronous.

7.1.8 `stop()`

Stop all components (Node/Controller) that is running are current process

7.1.9 `transfer_and_call(remote_process: str, func: Callable[..., Any], *args: Tuple[Any], **kwargs: Dict[Any, Any]) -> Union[Coroutine, Any]`

Send function along with arguments to execute on remote process. This method has some limitations. For example you should import modules inside function as modules might be unavailable on remote.

Parameters:

Name	Type	Description	Default
`remote_process`	`str`	Name of node where function should be executed	*required*
`args`	`Tuple[Any]`	Any positional arguments to execute function with.	`()`
`kwargs`	`Dict[Any, Any]`	Any keyword arguments to execute function with.	`{}`

Example

import os

from daffi import Global

async def get_remote_pid(): import os return os.getpid()

g = Global() g.transfer_and_call("node_name", get_remote_pid)

7.1.10 `wait_function(func_name: str) -> NoReturn`

Wait particular remote callback by name to be available. This method is useful when callback with the same name is registered on different nodes and you need at leas one node to be available.

Parameters:

Name	Type	Description	Default
`func_name`	`str`	Name of remote callback to wait.	*required*

7.1.11 `wait_function_async(func_name: str) -> NoReturn` `async`

Wait particular remote callback by name to be available (async version). This method is useful when callback with the same name is registered on different nodes and you need at leas one node to be available.

Parameters:

Name	Type	Description	Default
`func_name`	`str`	Name of remote callback to wait.	*required*

7.1.12 `wait_process(process_name: str) -> NoReturn`

Wait particular Node to be alive.

Parameters:

Name	Type	Description	Default
`process_name`	`str`	Name of `Node`	*required*

7.1.13 `wait_process_async(process_name: str) -> NoReturn` `async`

Wait particular Node to be alive (async version).

Parameters:

Name	Type	Description	Default
`process_name`	`str`	Name of `Node`	*required*

7.2 callback

Bases: `Decorator`

callback is decorator for registering remote callbacks from functions.

Example

from daffi.decorators import callback

@callback def my_func(*args,* *kwargs): ...

7.3 fetcher

Bases: `Decorator`

fetcher is decorator that converts a decorated method into a remote object call. Lets consider you have function with name and signature `my_awersome_function(a: int, b: int, c: str): ...` registered on one of nodes. Yo need to create fetcher with the same name and the same signature to call function on remote:

Example

from daffi.decorators import fetcher, **body_unknown**

@fetcher def my_awersome_function(a: int, b: int, c: str): # or use pass. Internal logic will be skipped in any case. only name and signature is important **body_unknown**(a, b, c)

7.3.1 Then we can call `my_awersome_function` on remote

7.3.2 !!! Execution modifier is binded to fetcher. No need to use `& FG` after execution.

result = my_awersome_function(1, 2, "abc")