Tavrida Documentation

Release 0.0.1

Sergey Bunatyan

April 25, 2019

Contents

CHAPTER 1

Source

https://github.com/sbunatyan/tavrida

CHAPTER 2

Contents

2.1 Installation

To install run:

```
pip install tavrida
```

2.2 Tutorial

2.2.1 Request Handling

Simple service that handles request

To implement simple service just follow the next steps:

1. Declare service controller
2. Define handler for some entry point (test_hello.hello)
3. Implement custom handler logic
4. Create configuration to connect to RabbitMQ
5. Create server that listens queue and publishes messages for given services
6. Start the server

Service Hello

```
from tavrida import config
from tavrida import dispatcher
from tavrida import server
from tavrida import service

@dispatcher.rpc_service("test_hello")
class HelloController(service.ServiceController):

    @dispatcher.rpc_method(service="test_hello", method="hello")
    def handler(self, request, proxy, param):
        print param
```

```
def run():

    creds = config.Credentials("guest", "guest")
    conf = config.ConnectionConfig("localhost", credentials=creds,
                                   async_engine=True)
    srv = server.Server(conf,
                        queue_name="test_service",
                        exchange_name="test_exchange",
                        service_list=[HelloController])
    srv.run()
```

To implement a client that makes calls to service use the following steps:

1. Create configuration to connect to RabbitMQ
2. Create discovery object. Discovery object is used to discover remote service's exchange by service name.
3. Create a client for a particular service. The source value is required and is useful for troubleshooting.
4. Make call to remote service. *Cast* function call is usual for client that lives outside RPC service. *Cast* means that you don't expect a response. As you make your call from some script, not from a service, you don't expect response.

Client to call Hello service

```
from tavrida import client
from tavrida import config
from tavrida import discovery
from tavrida import entry_point

creds = config.Credentials("guest", "guest")
conf = config.ConnectionConfig("localhost", credentials=creds)

disc = discovery.LocalDiscovery()
disc.register_remote_service(service_name="test_hello",
                             exchange_name="test_exchange")
cli = client.RPCClient(config=conf, service="test_hello", discovery=disc,
                       source="client_app")
cli.hello(param=123).cast()
```

2.2.2 Two services that intercommunicate (Request, Response, Error handling)

In this example we omit the client script as it's absolutely the same as in the previous example. Let's implement service *Hello* that handles request from some outer client, makes requests to the *World* service and handles responses and error messages from it.

1. Declare Hello service controller
2. Define handler for some entry point (test_hello.hello)
3. In this handler implement a call to the *World* service via proxy object. Give attention to that we use *call* method as we expect the response from *World* service.
4. Define handlers for response and error from remote entry point (test_world.world). Error handler always takes **only 2 parameters**: error message object and proxy object.
5. Create discovery object and register remote service (test_world). Discovery object is used to discover remote service's exchange by service name.

6. Bind discovery object to service controller.
7. Create configuration to connect to RabbitMQ
8. Create server that listens queue and publishes messages for given services
9. Start the server

Service Hello

```
from tavrida import config
from tavrida import dispatcher
from tavrida import server
from tavrida import service

@dispatcher.rpc_service("test_hello")
class HelloController(service.ServiceController):

    @dispatcher.rpc_method(service="test_hello", method="hello")
    def handler(self, request, proxy, param):
        print "---- request to hello ----"
        print param
        proxy.test_world.world(param=12345).call()

    @dispatcher.rpc_response_method(service="test_world", method="world")
    def world_resp(self, response, proxy, param):
        # Handles responses from test_world.world
        print "---- response from world to hello ----"
        print response.context
        print response.headers
        print param # == "world response"
        print "---------------------------------------"

    @dispatcher.rpc_error_method(service="test_world", method="world")
    def world_error(self, error, proxy):
        # Handles error from test_world.world
        print "---- error from hello ------"
        print error.context
        print error.headers
        print error.payload
        print "----------------------------"

def run():

    disc = discovery.LocalDiscovery()

    # register remote service's exchanges to send there requests (RPC calls)
    disc.register_remote_service("test_world", "test_world_exchange")
    HelloController.set_discovery(disc)

    # define connection parameters
    creds = config.Credentials("guest", "guest")
    conf = config.ConnectionConfig("localhost", credentials=creds,
                                   async_engine=True)
    # create server
    srv = server.Server(conf,
                        queue_name="test_service",
                        exchange_name="test_exchange",
```

```
                          service_list=[HelloController])
    srv.run()
```

Service World

Steps to implement the World service are pretty similar to the previous example. The only difference is remote service registration (test_hello) and binding the discovery object to service controller. In this example remote service registration is needed to send responses and error messages to test_hello service.

```
from tavrida import config
from tavrida import dispatcher
from tavrida import server
from tavrida import service

@dispatcher.rpc_service("test_world")
class WorldController(service.ServiceController):

    @dispatcher.rpc_method(service="test_world", method="world")
    def world(self, request, proxy, param):
        print "---- request to world------"
        print request.context
        print request.headers
        print param # == 12345
        print "--------------------------"
        return {"param": "world response"}

def run():

    disc = discovery.LocalDiscovery()

    # register remote service's exchange to send there requests,
    # responses, errors
    disc.register_remote_service("test_hello", "test_exchange")
    WorldController.set_discovery(disc)

    creds = config.Credentials("guest", "guest")
    conf = config.ConnectionConfig("localhost", credentials=creds)

    srv = server.Server(conf,
                        queue_name="test_world_service",
                        exchange_name="test_world_exchange",
                        service_list=[WorldController])
    srv.run()
```

2.2.3 Publication and Subscription

Hello Service (publisher)

1. Declare Hello service controller
2. In any request handler (or single script) use proxy to publish notification

```
from tavrida import config
from tavrida import dispatcher
from tavrida import server
```

```
from tavrida import service

@dispatcher.rpc_service("test_hello")
class HelloController(service.ServiceController):

    @dispatcher.rpc_method(service="test_hello", method="hello")
    def handler(self, request, proxy, param):
        print param
        proxy.publish(param="hello publication")

def run():

    # register service's notification exchange to publish notifications
    # Service 'test_hello' publishes notifications to it's exchange
    # 'test_notification_exchange'
    disc = discovery.LocalDiscovery()
    disc.register_local_publisher("test_hello",
                                  "test_notification_exchange")
    HelloController.set_discovery(disc)

    creds = config.Credentials("guest", "guest")
    conf = config.ConnectionConfig("localhost", credentials=creds,
                                   async_engine=True)
    srv = server.Server(conf,
                        queue_name="test_service",
                        exchange_name="test_exchange",
                        service_list=[HelloController])
    srv.run()
```

World service (subscriber)

1. Declare World service controller
2. Define subscription method

```
from tavrida import config
from tavrida import dispatcher
from tavrida import server
from tavrida import service

@dispatcher.rpc_service("test_world")
class WorldController(service.ServiceController):

    @dispatcher.subscription_method(service="test_hello", method="hello")
    def hello_subscription(self, notification, proxy, param):
        print "---- notification from hello ------"
        print param # == "hello publication"

def run():

    # register remote notification exchange to bind and get notifications
    # In this example service 'test_subscribe' gets notifications to it's queue
    # from 'test_notification_exchange' which is the publication exchange of
    # service 'test_hello'
    disc = discovery.LocalDiscovery()
    disc.register_remote_publisher("test_hello",
                                   "test_notification_exchange")
```

```
    WorldController.set_discovery(disc)

    creds = config.Credentials("guest", "guest")
    conf = config.ConnectionConfig("localhost", credentials=creds)

    srv = server.Server(conf,
                        queue_name="test_world_service",
                        exchange_name="test_world_exchange",
                        service_list=[WorldController])
    srv.run()
```

2.3 Controller

2.3.1 Controller class

To declare controller class you should inherit it from `tavrida.service.ServiceController` and decorate with `tavrida.dispatcher.rpc_service()` decorator

```
from tavrida import dispatcher
from tavrida import service

@dispatcher.rpc_service("test_hello")
class HelloController(service.ServiceController):
    pass
```

Request handler can return 2 type of responses (dict, `tavrida.messages.Response`, `tavrida.messages.Error`) or None value.

If dict is returned it will be converted to the `tavrida.messages.Response` object. If any exception raises during message processing it is converted to `tavrida.messages.Error` object.

But you can, of course, return `tavrida.messages.Response` or `tavrida.messages.Error` object explicitly.

Controller is instantiated on `tavrida.server.Server` start. Each service controller class owns a `tavrida.dispatcher.Dispatcher` and discovery object.

If you are planning to execute calls from any of the handlers you should bind discovery object to the service class before `tavrida.server.Server` starts.

```
disc.register_remote_service("remote_service", "remote_service_exchange")
HelloController.set_discovery(disc)
```

2.3.2 Handlers

Handlers are methods of service controllers that are called on message (`tavrida.messages.IncomingRequest` request, `tavrida.messages.IncomingResponse` response, `tavrida.messages.IncomingError` error, `tavrida.messages.IncomingNotification` notification) arrival.

Each handler is bound to `tavrida.entry_point.EntryPoint` which can be considered as an address to deliver the message.

Each handler receives two parameters (at first two positions): message and `tavrida.proxies.RPCProxy`. Class of incoming message depends on handler type. Using `tavrida.proxies.RPCProxy` object you can execute calls to remote services. In such calls the service's discovery object is used.

All following parameters are custom parameters of a particular method. In the following example *param* is such parameter.

Request handler

`tavrida.messages.IncomingRequest` routing is based on the name of local service entry point. For example for *test_hello* service the correct entry point service value is *test_hello*

```
@dispatcher.rpc_method(service="test_hello", method="hello")
def handler(self, request, proxy, param):
    return {"parameter": "value"}
```

Response handler

`tavrida.messages.IncomingResponse` routing is based on the name of remote service entry point. For example for *test_hello* service and remote entry point *remote_service.remote_method* the correct entry point value is *remote_service.remote_method*

```
@dispatcher.rpc_response_method(service="remote_service", method="remote_method")
def world_resp(self, response, proxy, param):
    pass
```

Error handler

`tavrida.messages.IncomingError` routing is based on the name of remote service entry point. For example for *test_hello* service and remote entry point *remote_service.remote_method* the correct entry point value is *remote_service.remote_method* Error handler takes strictly **two** parameters. The first (error) parameter has a property *payload* that is a dict of 3 keys: *class*, *message*, *name*. All these keys are mapped to string values.

```
@dispatcher.rpc_error_method(service="remote_service", method="remote_method")
def world_error(self, error, proxy):
    pass
```

Subscription handler

`tavrida.messages.IncomingNotification` routing is based on the name of remote publisher entry point. Such entry point can be considered as notification topic. For example for *test_hello* service and remote entry point *remote_service.remote_method* the correct entry point value is *remote_service.remote_method*

```
@dispatcher.subscription_method(service="remote_service", method="remote_method")
def hello_subscription(self, notification, proxy, param):
    pass
```

Resulting code example

```
from tavrida import dispatcher
from tavrida import service

@dispatcher.rpc_service("test_hello")
class HelloController(service.ServiceController):

    @dispatcher.rpc_method(service="test_hello", method="hello")
```

```
    def handler(self, request, proxy, param):
        return {"parameter": "value"}

    @dispatcher.rpc_response_method(service="remote_service", method="remote_method")
    def world_resp(self, response, proxy, param):
        pass

    @dispatcher.rpc_error_method(service="remote_service", method="remote_method")
    def world_error(self, error, proxy):
        pass

    @dispatcher.subscription_method(service="remote_service", method="remote_method")
    def hello_subscription(self, notification, proxy, param):
        pass
```

2.4 Messages

Messages that are used in handlers are of two types: Incoming and Outgoing

2.4.1 Incoming messages

There 4 types of incoming messages: `tavrida.messages.IncomingRequest`, `tavrida.messages.IncomingResponse`, `tavrida.messages.IncomingError`, `tavrida.messages.IncomingNotification`

All messages have properties:

Property	Type	Description
correlation_id	string	Unique identifier of remote service calls chain.
request_id	string	Unique identifier of pair request - response/error
message_id	string	Unique single message identifier
message_type	string	Message type: request, response, error, notification
reply_to	`tavrida.entry_point.EntryPoint`	Entry point to send response/error to
source	`tavrida.entry_point.EntryPoint`	Entry point of the message source
destination	`tavrida.entry_point.EntryPoint`	Entry point of the message destination
headers	dict	dict of message headers (properties above)
context	dict	dict if context values
payload	dict	dict of incoming handler parameters

Notes

correlation_id - Unique identifier of the chain of calls between multiple services. For example: A <-> B <-> C -> D.

request_id - Unique identifier of the pair of messages request - response/error between 2 services. For example: A <-> B.

headers - dict of the properties above (named headers). It is mutable unlike properties. But headers and properties are not synchronized.

context - additional data that can be used in handlers. By default contains payload data and is updated at each hop. That means that if you have a chain if requests, context will be updated with all incoming parameters of each handler. As it is a simple dict the conflicting keys will be overwritten.

2.4.2 Outgoing messages

There 4 types of incoming messages: `tavrida.messages.Request`, `tavrida.messages.Response`, `tavrida.messages.Error`, `tavrida.messages.Notification` and their structure is the same as for incoming messages.

2.4.3 Under the hood

Messages are transported via RabbitMQ. Message headers are fair RabbitMQ headers: correlation_id, request_id, message_id, message_type, reply_to, source, source, destination.

Message payload is a valid JSON object that consists of 2 sub-objects:

```
{
    "context": {"some_key": "some_value"},
    "payload": {"parameter": "value"}
}
```

context holds arbitrary values. By default it is filled with the payload values and is updated after each request. That means that if you have a chain of 2 calls: service A -> service B -> service C, context will hold incoming parameters for both calls. But if at any hop parameter names are equal, the old value is overwritten by the new one. Actually context is just a python dict that is updated with "update" method.

payload holds custom parameters that defined in handler. Names of payload keys should be equal to names of handler parameters. If you have a handler:

```
@dispatcher.rpc_method(service="test_hello", method="hello")
def handler(self, request, proxy, param1, param1):
    return {"param3": "value3"}
```

your payload should look like:

```
"payload": {"param1": "value1", "param2": "value2"}
```

2.5 Middlewares

Tavrida has two types of middlewares: incoming and outgoing. They are executed just before (after) the handler call.

Middleware is a simple class inherited from `tavrida.middleware.Middleware` that implements a single method `process()`

This method takes message and returns the result message for the next middleware.

When you add middleware it is placed in the list and therefore the addition order is significant.

2.5.1 Incoming Middlewares

Example how to add incoming middleware:

```
from tavrida import dispatcher
from tavrida import middleware
from tavrida import service

class MyMiddleware(middleware.Middleware):
    def process(self, message):
        print "middleware call"
        return message

@dispatcher.rpc_service("test_hello")
class HelloController(service.ServiceController):

    def __init__(self, postprocessor):
        super(self, HelloController).__init__(postprocessor)
        self.add_incoming_middleware(MyMiddleware())

    @dispatcher.rpc_method(service="test_hello", method="hello")
    def handler(self, request, proxy, param):
        return {"parameter": "value"}
```

By default incoming message takes `tavrida.messages.IncomingRequest`.

Incoming middleware can return 3 type of messages

- If IncomingRequest is returned, it will be passed to the next middleware in the list.
- If Response/Error is returned, the chain of middleware processing is broken and message is returned to the calling service (of course if the incoming message is of type IncomingRequestCall)
- If any other type of response is returned the exception raises.

If exception is raised in middleware it is automatically converted to `tavrida.messages.Error`

2.5.2 Outgoing Middlewares

Outgoing middlewares are **NOT** called while executing call via proxy object.

Example how to add outgoing middleware:

```
from tavrida import dispatcher
from tavrida import middleware
from tavrida import service

class MyMiddleware(middleware.Middleware):
    def process(self, message):
        print "middleware call"
        return message

@dispatcher.rpc_service("test_hello")
class HelloController(service.ServiceController):

    def __init__(self, postprocessor):
        super(self, HelloController).__init__(postprocessor)
        self.add_outgoing_middleware(MyMiddleware())

    @dispatcher.rpc_method(service="test_hello", method="hello")
```

```
18      def handler(self, request, proxy, param):
19          return {"parameter": "value"}
```

By default outgoing middleware takes `tavrida.messages.Response` or `tavrida.messages.Error` message.

The result value of outgoing middleware should be of the same type. Otherwise exception is raised.

If exception is raised in outgoing middleware the message processing is stopped.

2.6 Discovery

Discovery object is used to discover remote service exchange to send messages to.

2.6.1 Discovery types

It holds 3 types of pairs *service_name:service_exchange*:

1. For remote service. Is used to send requests, responses, errors to remote service.
2. For remote service publisher. Is used to subscribe for notifications from remote service.
3. For local service publisher. Is used to publish notifications by local service.

Discovery service (.ds) config file

To use .ds file you need to describe services in .ds file in format

```
[service1]
exchange=service1_exchange
notifications=service1_notifications

[service2]
exchange=service2_exchange
notifications=service2_notifications

[service3]
exchange=service3_exchange
notifications=service3_notifications

[service4]
exchange=service4_exchange
```

And then to load .ds file to `tavrida.discovery.FileBasedDiscoveryService`

```
1  from tavrida import discovery
2
3  srv1_disc = discovery.FileBasedDiscoveryService("services.ds", "service1")
```

Discovery without .ds file

To register all types of services use `tavrida.discovery.LocalDiscovery`:

```
from tavrida import discovery

disc = discovery.LocalDiscovery()

# register remote service's exchange to send equests,
# responses, errors
disc.register_remote_service("remote_service", "remote_service_exchange")

# register service notification exchange to publish notifications
# Service 'local_service' publishes notifications to its exchange
# 'local_service_exchange'
disc.register_local_publisher("local_service", "local_service_exchange")

# register remote notification exchange to bind to and get notifications
# In this example service 'local_service' gets notifications to it's queue
# from 'remote_notifications_exchange' which is the publication exchange of
# service 'remote_Service'
disc.register_remote_publisher("remote_service", "remote_notifications_exchange")
```

2.6.2 Discovery binding

Before server starts each service that needs to interact with other service should be binded to one discovery object.

Therefore if you have multiple services and subsequently multiple discovery objects you should register each required remote or local service in corresponding discovery service.

```
from tavrida import discovery

disc = discovery.LocalDiscovery()
disc.register_remote_service("remote_service", "remote_service_exchange")
MyServiceController.set_discovery(disc)
```

2.6.3 Discovery for proxy

Besides that you should provide discovery object while creation `tavrida.client.RPCClient` object.

```
from tavrida import client
from tavrida import discovery

disc = discovery.LocalDiscovery()
disc.register_remote_service(service_name="remote_service",
                             exchange_name="remote_exchange")
cli = client.RPCClient(config=conf, service="test_hello", discovery=disc,
                       source=source)
```

Soon the discovery that uses central settings storage will be implemented. But you can implement your own discovery class. The only demand is to inherit it from `tavrida.discovery.AbstractDiscovery`

2.7 Client

To execute calls from third-party applications use `tavrida.client.RPCClient` object.

2.7.1 Client parameters

- You can pass optional *correlation_id* parameter. If remote service executes the subsequent call to the next service *correlation_id* will passed.
- You can pass additional *header* parameters to the remote service.

There are several ways to create and use client.

```
from tavrida import client
from tavrida import config
from tavrida import discovery
from tavrida import entry_point

creds = config.Credentials("guest", "guest")
conf = config.ConnectionConfig("localhost", credentials=creds)

# You should provide discovery service object to client
disc = discovery.LocalDiscovery()
disc.register_remote_service(service_name="test_hello",
                             exchange_name="test_exchange")

cli = client.RPCClient(config=conf, discovery=disc, source=source)
cli.test_hello.hello(param=123).cast(correlation_id="123-456")
```

```
from tavrida import client
from tavrida import config
from tavrida import discovery
from tavrida import entry_point

creds = config.Credentials("guest", "guest")
conf = config.ConnectionConfig("localhost", credentials=creds)

# You should provide discovery service object to client
disc = discovery.LocalDiscovery()
disc.register_remote_service(service_name="test_hello",
                             exchange_name="test_exchange")

# If you want to provide source as a string
cli = client.RPCClient(config=conf, discovery=disc, source="source_service")
cli.test_hello.hello(param=123).cast(correlation_id="123-456")

cli = client.RPCClient(config=conf, discovery=disc, source="source.method")
cli.test_hello.hello(param=123).cast(correlation_id="123-456")
```

2.8 Proxy

Proxy is the object that allows you to execute calls to remote services and publish notifications.

Each handler gets proxy object as a second parameter.

2.8.1 Proxy parameters

- You can pass optional *correlation_id* parameter.

If remote service executes the second call to the next service correlation_id will be the same.

- To the `call()` or `cast()` method you can pass *correlation_id*, *context*, *source* values.
- To the `call()` method you can provide *reply_to* parameter.
- You can add header parameter to the proxy using *add_headers* method

```
from tavrida import dispatcher
from tavrida import service

@dispatcher.rpc_service("local_service")
class LocalServiceController(service.ServiceController):

    @dispatcher.rpc_method(service="local_service", method="rpc_method")
    def rpc_method(self, request, proxy, param):
        proxy.remote_service.method(value="call-1").call(correlation_id="123=456)
        proxy.remote_service.method(value="call-1").cast()
        proxy.publish(value="notification_value")
```

2.9 Configuration

2.9.1 Config parameters

Tavrida configuration passes most of the parameters to pika. Information about them you can find here.

The Tavrida specific configuration parameters are:

- *reconnect_attempts* (Int) - number of attempts to reconnect to RabbitMQ on failure. The negative value means **infinite** number. The **default** is **-1**.
- *async_engine* (Bool) - use pika SelectConnection. It is more productive but less tested. By **default** is **False**.

Example:

```
from tavrida import config

creds = config.Credentials("guest", "guest")
conf = config.ConnectionConfig(host="localhost",
                               credentials=creds,
                               port=5672,
                               reconnect_attempts=3,
                               async_engine=True)
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

CHAPTER 4

Brief service example

Hello service

```
from tavrida import config
from tavrida import dispatcher
from tavrida import server
from tavrida import service

@dispatcher.rpc_service("test_hello")
class HelloController(service.ServiceController):

    @dispatcher.rpc_method(service="test_hello", method="hello")
    def handler(self, request, proxy, param):
        print param

def run():

    creds = config.Credentials("guest", "guest")
    conf = config.ConnectionConfig("localhost", credentials=creds,
                                   async_engine=True)
    srv = server.Server(conf,
                        queue_name="test_service",
                        exchange_name="test_exchange",
                        service_list=[HelloController])
    srv.run()
```