
Tavrida Documentation

Release 0.0.1

Sergey Bunatyan

April 25, 2019

1	Source	1
2	Contents	3
2.1	Installation	3
2.2	Tutorial	3
2.3	Controller	8
2.4	Messages	10
2.5	Middlewares	11
2.6	Discovery	13
2.7	Client	14
2.8	Proxy	15
2.9	Configuration	16
3	Indices and tables	17
4	Brief service example	19

Source

<https://github.com/sbunatyan/tavrida>

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

```
1 from tavrida import config
2 from tavrida import dispatcher
3 from tavrida import server
4 from tavrida import service
5
6 @dispatcher.rpc_service("test_hello")
7 class HelloController(service.ServiceController):
8
9     @dispatcher.rpc_method(service="test_hello", method="hello")
10     def handler(self, request, proxy, param):
11         print param
```

```
12
13 def run() :
14
15     creds = config.Credentials("guest", "guest")
16     conf = config.ConnectionConfig("localhost", credentials=creds,
17                                   async_engine=True)
18
19     srv = server.Server(conf,
20                         queue_name="test_service",
21                         exchange_name="test_exchange",
22                         service_list=[HelloController])
23
24     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

```
1 from tavrida import client
2 from tavrida import config
3 from tavrida import discovery
4 from tavrida import entry_point
5
6 creds = config.Credentials("guest", "guest")
7 conf = config.ConnectionConfig("localhost", credentials=creds)
8
9 disc = discovery.LocalDiscovery()
10 disc.register_remote_service(service_name="test_hello",
11                             exchange_name="test_exchange")
12 cli = client.RPCClient(config=conf, service="test_hello", discovery=disc,
13                       source="client_app")
14 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

```

1 from tavrida import config
2 from tavrida import dispatcher
3 from tavrida import server
4 from tavrida import service
5
6 @dispatcher.rpc_service("test_hello")
7 class HelloController(service.ServiceController):
8
9     @dispatcher.rpc_method(service="test_hello", method="hello")
10    def handler(self, request, proxy, param):
11        print "---- request to hello ----"
12        print param
13        proxy.test_world.world(param=12345).call()
14
15    @dispatcher.rpc_response_method(service="test_world", method="world")
16    def world_resp(self, response, proxy, param):
17        # Handles responses from test_world.world
18        print "---- response from world to hello ----"
19        print response.context
20        print response.headers
21        print param # == "world response"
22        print "-----"
23
24    @dispatcher.rpc_error_method(service="test_world", method="world")
25    def world_error(self, error, proxy):
26        # Handles error from test_world.world
27        print "---- error from hello ----"
28        print error.context
29        print error.headers
30        print error.payload
31        print "-----"
32
33 def run():
34
35     disc = discovery.LocalDiscovery()
36
37     # register remote service's exchanges to send there requests (RPC calls)
38     disc.register_remote_service("test_world", "test_world_exchange")
39     HelloController.set_discovery(disc)
40
41     # define connection parameters
42     creds = config.Credentials("guest", "guest")
43     conf = config.ConnectionConfig("localhost", credentials=creds,
44                                   async_engine=True)
45
46     # create server
47     srv = server.Server(conf,
48                         queue_name="test_service",
49                         exchange_name="test_exchange",

```

```
49         service_list=[HelloController])
50     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.

```
1  from tavrida import config
2  from tavrida import dispatcher
3  from tavrida import server
4  from tavrida import service
5
6  @dispatcher.rpc_service("test_world")
7  class WorldController(service.ServiceController):
8
9      @dispatcher.rpc_method(service="test_world", method="world")
10     def world(self, request, proxy, param):
11         print "---- request to world-----"
12         print request.context
13         print request.headers
14         print param # == 12345
15         print "-----"
16         return {"param": "world response"}
17
18     def run():
19
20         disc = discovery.LocalDiscovery()
21
22         # register remote service's exchange to send there requests,
23         # responses, errors
24         disc.register_remote_service("test_hello", "test_exchange")
25         WorldController.set_discovery(disc)
26
27         creds = config.Credentials("guest", "guest")
28         conf = config.ConnectionConfig("localhost", credentials=creds)
29
30         srv = server.Server(conf,
31                             queue_name="test_world_service",
32                             exchange_name="test_world_exchange",
33                             service_list=[WorldController])
34         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

```
1  from tavrida import config
2  from tavrida import dispatcher
3  from tavrida import server
```

```

4 from tavrida import service
5
6 @dispatcher.rpc_service("test_hello")
7 class HelloController(service.ServiceController):
8
9     @dispatcher.rpc_method(service="test_hello", method="hello")
10    def handler(self, request, proxy, param):
11        print param
12        proxy.publish(param="hello publication")
13
14    def run():
15
16        # register service's notification exchange to publish notifications
17        # Service 'test_hello' publishes notifications to it's exchange
18        # 'test_notification_exchange'
19        disc = discovery.LocalDiscovery()
20        disc.register_local_publisher("test_hello",
21                                    "test_notification_exchange")
22        HelloController.set_discovery(disc)
23
24        creds = config.Credentials("guest", "guest")
25        conf = config.ConnectionConfig("localhost", credentials=creds,
26                                     async_engine=True)
27
28        srv = server.Server(conf,
29                            queue_name="test_service",
30                            exchange_name="test_exchange",
31                            service_list=[HelloController])
32
33        srv.run()

```

World service (subscriber)

1. Declare World service controller
2. Define subscription method

```

1 from tavrida import config
2 from tavrida import dispatcher
3 from tavrida import server
4 from tavrida import service
5
6 @dispatcher.rpc_service("test_world")
7 class WorldController(service.ServiceController):
8
9     @dispatcher.subscription_method(service="test_hello", method="hello")
10    def hello_subscription(self, notification, proxy, param):
11        print "---- notification from hello ----"
12        print param # == "hello publication"
13
14    def run():
15
16        # register remote notification exchange to bind and get notifications
17        # In this example service 'test_subscribe' gets notifications to it's queue
18        # from 'test_notification_exchange' which is the publication exchange of
19        # service 'test_hello'
20        disc = discovery.LocalDiscovery()
21        disc.register_remote_publisher("test_hello",
22                                     "test_notification_exchange")

```

```
23 WorldController.set_discovery(disc)
24
25 creds = config.Credentials("guest", "guest")
26 conf = config.ConnectionConfig("localhost", credentials=creds)
27
28 srv = server.Server(conf,
29                     queue_name="test_world_service",
30                     exchange_name="test_world_exchange",
31                     service_list=[WorldController])
32
33 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

```
1 from tavrida import dispatcher
2 from tavrida import service
3
4 @dispatcher.rpc_service("test_hello")
5 class HelloController(service.ServiceController):
6     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.

```
1 disc.register_remote_service("remote_service", "remote_service_exchange")
2 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*

```
1 @dispatcher.rpc_method(service="test_hello", method="hello")
2 def handler(self, request, proxy, param):
3     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*

```
1 @dispatcher.rpc_response_method(service="remote_service", method="remote_method")
2 def world_resp(self, response, proxy, param):
3     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.

```
1 @dispatcher.rpc_error_method(service="remote_service", method="remote_method")
2 def world_error(self, error, proxy):
3     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*

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

Resulting code example

```
1 from tavrida import dispatcher
2 from tavrida import service
3
4 @dispatcher.rpc_service("test_hello")
5 class HelloController(service.ServiceController):
6
7     @dispatcher.rpc_method(service="test_hello", method="hello")
```

```
8     def handler(self, request, proxy, param):
9         return {"parameter": "value"}
10
11     @dispatcher.rpc_response_method(service="remote_service", method="remote_method")
12     def world_resp(self, response, proxy, param):
13         pass
14
15     @dispatcher.rpc_error_method(service="remote_service", method="remote_method")
16     def world_error(self, error, proxy):
17         pass
18
19     @dispatcher.subscription_method(service="remote_service", method="remote_method")
20     def hello_subscription(self, notification, proxy, param):
21         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
<code>correlation_id</code>	string	Unique identifier of remote service calls chain.
<code>request_id</code>	string	Unique identifier of pair request - response/error
<code>message_id</code>	string	Unique single message identifier
<code>message_type</code>	string	Message type: request, response, error, notification
<code>reply_to</code>	<code>tavrida.entry_point.EntryPoint</code>	Entry point to send response/error to
<code>source</code>	<code>tavrida.entry_point.EntryPoint</code>	Entry point of the message source
<code>destination</code>	<code>tavrida.entry_point.EntryPoint</code>	Entry point of the message destination
<code>headers</code>	dict	dict of message headers (properties above)
<code>context</code>	dict	dict if context values
<code>payload</code>	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 of 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:

```
1 from tavrida import dispatcher
2 from tavrida import middleware
3 from tavrida import service
4
5 class MyMiddleware(middleware.Middleware):
6     def process(self, message):
7         print "middleware call"
8         return message
9
10 @dispatcher.rpc_service("test_hello")
11 class HelloController(service.ServiceController):
12
13     def __init__(self, postprocessor):
14         super(self, HelloController).__init__(postprocessor)
15         self.add_incoming_middleware(MyMiddleware())
16
17     @dispatcher.rpc_method(service="test_hello", method="hello")
18     def handler(self, request, proxy, param):
19         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:

```
1 from tavrida import dispatcher
2 from tavrida import middleware
3 from tavrida import service
4
5 class MyMiddleware(middleware.Middleware):
6     def process(self, message):
7         print "middleware call"
8         return message
9
10 @dispatcher.rpc_service("test_hello")
11 class HelloController(service.ServiceController):
12
13     def __init__(self, postprocessor):
14         super(self, HelloController).__init__(postprocessor)
15         self.add_outgoing_middleware(MyMiddleware())
16
17     @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`:

```
1 from tavrida import discovery
2
3 disc = discovery.LocalDiscovery()
4
5 # register remote service's exchange to send equests,
6 # responses, errors
7 disc.register_remote_service("remote_service", "remote_service_exchange")
8
9 # register service notification exchange to publish notifications
10 # Service 'local_service' publishes notifications to its exchange
11 # 'local_service_exchange'
12 disc.register_local_publisher("local_service", "local_service_exchange")
13
14 # register remote notification exchange to bind to and get notifications
15 # In this example service 'local_service' gets notifications to it's queue
16 # from 'remote_notifications_exchange' which is the publication exchange of
17 # service 'remote_Service'
18 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.

```
1 from tavrida import discovery
2
3 disc = discovery.LocalDiscovery()
4 disc.register_remote_service("remote_service", "remote_service_exchange")
5 MyServiceController.set_discovery(disc)
```

2.6.3 Discovery for proxy

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

```
1 from tavrida import client
2 from tavrida import discovery
3
4 disc = discovery.LocalDiscovery()
5 disc.register_remote_service(service_name="remote_service",
6                             exchange_name="remote_exchange")
7 cli = client.RPCClient(config=conf, service="test_hello", discovery=disc,
8                        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 be passed.
- You can pass additional *header* parameters to the remote service.

There are several ways to create and use client.

```

1 from tavrida import client
2 from tavrida import config
3 from tavrida import discovery
4 from tavrida import entry_point
5
6 creds = config.Credentials("guest", "guest")
7 conf = config.ConnectionConfig("localhost", credentials=creds)
8
9 # You should provide discovery service object to client
10 disc = discovery.LocalDiscovery()
11 disc.register_remote_service(service_name="test_hello",
12                             exchange_name="test_exchange")
13
14 cli = client.RPCClient(config=conf, discovery=disc, source=source)
15 cli.test_hello.hello(param=123).cast(correlation_id="123-456")

```

```

1 from tavrida import client
2 from tavrida import config
3 from tavrida import discovery
4 from tavrida import entry_point
5
6 creds = config.Credentials("guest", "guest")
7 conf = config.ConnectionConfig("localhost", credentials=creds)
8
9 # You should provide discovery service object to client
10 disc = discovery.LocalDiscovery()
11 disc.register_remote_service(service_name="test_hello",
12                             exchange_name="test_exchange")
13
14 # If you want to provide source as a string
15 cli = client.RPCClient(config=conf, discovery=disc, source="source_service")
16 cli.test_hello.hello(param=123).cast(correlation_id="123-456")
17
18 cli = client.RPCClient(config=conf, discovery=disc, source="source.method")
19 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

```
1 from tavrida import dispatcher
2 from tavrida import service
3
4 @dispatcher.rpc_service("local_service")
5 class LocalServiceController(service.ServiceController):
6
7     @dispatcher.rpc_method(service="local_service", method="rpc_method")
8     def rpc_method(self, request, proxy, param):
9         proxy.remote_service.method(value="call-1").call(correlation_id="123=456")
10        proxy.remote_service.method(value="call-1").cast()
11        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:

```
1 from tavrida import config
2
3 creds = config.Credentials("guest", "guest")
4 conf = config.ConnectionConfig(host="localhost",
5                                credentials=creds,
6                                port=5672,
7                                reconnect_attempts=3,
8                                async_engine=True)
```

Indices and tables

- `genindex`
- `modindex`
- `search`

Brief service example

Hello service

```
1 from tavrida import config
2 from tavrida import dispatcher
3 from tavrida import server
4 from tavrida import service
5
6 @dispatcher.rpc_service("test_hello")
7 class HelloController(service.ServiceController):
8
9     @dispatcher.rpc_method(service="test_hello", method="hello")
10    def handler(self, request, proxy, param):
11        print param
12
13 def run():
14
15     creds = config.Credentials("guest", "guest")
16     conf = config.ConnectionConfig("localhost", credentials=creds,
17                                   async_engine=True)
18
19     srv = server.Server(conf,
20                         queue_name="test_service",
21                         exchange_name="test_exchange",
22                         service_list=[HelloController])
23
24     srv.run()
```