wampy Documentation

Release 0.9.19

simon harrison

Dec 21, 2018

Contents

WAMP RPC and Pub/Sub for your Python apps and microservices

This is a Python implementation of **WAMP** not requiring Twisted or asyncio, enabling use within classic blocking Python applications. It is a light-weight alternative to autobahn.

With **wampy** you can quickly and easily create your own **WAMP** clients, whether this is in a web app, a microservice, a script or just in a Python shell.

wampy tries to provide an intuitive API for your **WAMP** messaging.

CHAPTER 1

WAMP

Background to the Web Application Messaging Protocol of which wampy implements.

What is WAMP?

The WAMP Protocol is a powerful tool for your web applications and microservices - else just for your free time, fun and games!

WAMP facilitates communication between independent applications over a common "router". An actor in this process is called a **Peer**, and a **Peer** is either a **Client** or the **Router**.

WAMP messaging occurs between **Clients** over the **Router** via **Remote Procedure Call (RPC)** or the **Publish/Subscribe** pattern. As long as your **Client** knows how to connect to a **Router** it does not then need to know anything further about other connected **Peers** beyond a shared string name for an endpoint or **Topic**, i.e. it does not care where another **Client** application is, how many of them there might be, how they might be written or how to identify them. This is more simple than other messaging protocols, such as AMQP for example, where you also need to consider exchanges and queues in order to explicitly connect to other actors from your applications.

WAMP is most commonly a WebSocket subprotocol (runs on top of WebSocket) that uses JSON as message serialization format. However, the protocol can also run with MsgPack as serialization, run over raw TCP or in fact any message based, bidirectional, reliable transport - but **wampy** (currently) runs over websockets only.

For further reading please see some of the popular blog posts on WAMP such as http://tavendo.com/blog/post/is-crossbar-the-future-of-python-web-apps/.

CHAPTER 2

User Guide

Running a wampy application or interacting with any other WAMP application

What is wampy?

This is a Python implementation of **WAMP** not requiring Twisted or asyncio, enabling use within classic blocking Python applications. It is a light-weight alternative to autobahn.

With **wampy** you can quickly and easily create your own **WAMP** clients, whether this is in a web app, a microservice, a script or just in a Python shell.

wampy tries to provide an intuitive API for your **WAMP** messaging.

A wampy Application

This is a fully fledged example of a wampy application that implements all 4 WAMP Roles: caller, callee, publisher and subscriber.

```python
from wampy.peers.clients import Client
from wampy.roles import callee
from wampy.roles import subscriber


class WampyApp(Client):

    @callee
    def get_weather(self, *args, **kwargs):
        weather = self.call("another.example.app.endpoint")
        return weather

    @subscriber(topic="global-weather")
    def weather_events(self, weather_data):
        # process weather data here
        self.publish(topic="wampy-weather", message=weather_data)
```

Here the method decorated by @callee is a callable remote procedure. In this example, it also acts as a Caller, by calling another remote procedure and then returning the result.

And the method decorated by @subscribe implements the Subscriber Role, and when it receives an Event it then acts as a Publisher, and publishes a new message to a topic.

Note that the `call` and `publish` APIs are provided by the super class, `Client`.

Running The Application

wampy provides a command line interface tool to start the application.

```
$ wampy run path.to.your.module.including.module_name:WampyApp
```

For example, running one of the **wampy** example applications.

```
$ wampy run docs.examples.services:BinaryNumberService --config './wampy/testing/
↪configs/crossbar.config.ipv4.json'
```

A wampy Client

If you're working from a Python shell or script you can connect to a Router as follows.

1. Router is running on localhost, port 8080, start and stop manually.

```
from wampy.peers import Client

client = Client()
client.start()  # connects to the Router & establishes a WAMP Session
# send some WAMP messages here
client.stop()  # ends Session, disconnects from Router
```

2. Router is running on localhost, port 8080, context-manage the Session

```
from wampy.peers import Client

with Client() as client:
    # send some WAMP messages here

# on exit, the Session and connection are gracefully closed
```

3. Router is on example.com, port 8080, context-managed client again

```
from wampy.peers import Client

with Client(url="ws://example.com:8080") as client:
    # send some WAMP messages here

# exits as normal
```

Under the hood wampy creates an instance of a Router representaion because a Session is a managed conversation between two Peers - a Client and a Router. Because wampy treats a Session like this, there is actually also a *fourth* method of connection, as you can create the Router instance yourself and pass this into a Client directly. This is bascically only useful for test and CI environments, or local setups during development, or for fun. See the wampy tests for examples and the wampy wrapper around the Crossbar.io Router.

Sending a Message

When a **wampy** client starts up it will send the **HELLO** message for you and begin a **Session**. Once you have the **Session** you can construct and send a **WAMP** message yourself, if you so choose. But **wampy** has the `publish` and `rpc` APIs so you don't have to.

But if you did want to do it yourself, here's an example how to...

Given a **Crossbar.io** server running on localhost on port 8080, a **realm** of "realm1", and a remote procedure "foobar", send a **CALL** message with **wampy** as follows:

```
In [1]: from wampy.peers.clients import Client

In [2]: from wampy.messages.call import Call

In [3]: client = Client()

In [4]: message = Call(procedure="foobar", args=(), kwargs={})

In [5]: with client:
            client.send_message(message)
```

This example assumes a Router running on localhost and a second Peer attached over the same realm who hjas registered the callee "foobar"

Note that in the example, as you leave the context managed function call, the client will send a **GOODBYE** message and your **Session** will end.

wampy does not want you to waste time constructing messages by hand, so the above can be replaced with:

```
In [1]: from wampy.peers.clients import Client

In [2]: client = Client()

In [5]: with client:
            client.rpc.foobar(*args, **kwargs)
```

Under the hood, **wampy** has the `RpcProxy` object that implements the `rpc` API.

Publishing to a Topic

To publish to a topic you simply call the `publish` API on any wampy client with the topic name and message to deliver.

```
from wampy.peers.clients import Client
from wampy.peers.routers import Crossbar

with Client(router=Crossbar()) as client:
    client.publish(topic="foo", message={'foo': 'bar'})
```

The message can be whatever JSON serializable object you choose.

Note that the Crossbar router does require a path to an expected `config.yaml`, but here a default value is used. The default for Crossbar is `"./crossbar/config.json"`.

Subscribing to a Topic

You need a long running wampy application process for this.

```
from wampy.peers.clients import Client
from wampy.roles.subscriber import subscribe


class WampyApp(Client):

    @subscribe(topic="topic-name")
    def weather_events(self, topic_data):
        # do something with the ``topic_data`` here
        pass
```

See runnning a wampy application for executing the process.

Remote Procedure Calls

Classic

Conventional remote procedure calling over Crossbar.io.

```
from wampy.peers import Client
from wampy.peers.routers import Crossbar

with Client(router=Crossbar()) as client:
    result = client.call("example.app.com.endpoint", *args, **kwargs)
```

Microservices

Inspired by the nameko project.

```
from wampy.peers import Client
from wampy.peers.routers import Crossbar

with Client(router=Crossbar()) as client:
    result = client.rpc.endpoint(**kwargs)
```

See nameko_wamp for usage.

Exception Handling

When calling a remote procedure an `Exception` might be raised by the remote application. It this happens the *Callee's* `Exception` will be wrapped in a wampy `RemoteError` and will contain the name of the remote procedure that raised the error, the `request_id`, the exception type and any message.

```
from wampy.errors import RemoteError
from wampy.peers.clients import Client
```

```python
with Client() as client:

    try:
        response = client.rpc.some_unreliable_procedure()
    except RemoteError as rmt_err:
        # do stuff here to recover from the error or
        # fail gracefully
```

Authentication Methods

The Realm is a WAMP routing and administrative domain, optionally protected by authentication and authorization.

In the WAMP Basic Profile without session authentication the Router will reply with a "WELCOME" or "ABORT" message.

```
,------.          ,------.
|Client|          |Router|
`--+---'          `--+---'
   |      HELLO      |
   | ---------------->
   |                 |
   |     WELCOME     |
   | <----------------
,--+---.          ,--+---.
|Client|          |Router|
`------'          `------'
```

The Advanced router Profile provides some authentication options at the WAMP level - although your app may choose to use transport level auth (e.g. cookies or TLS certificates) or implement its own system (e.g. on the remote procedure).

```
,------.          ,------.
|Client|          |Router|
`--+---'          `--+---'
   |      HELLO      |
   | ---------------->
   |                 |
   |    CHALLENGE    |
   | <----------------
   |                 |
   |   AUTHENTICATE  |
   | ---------------->
   |                 |
   | WELCOME or ABORT|
   | <----------------
,--+---.          ,--+---.
|Client|          |Router|
`------'          `------'
```

Challenge Response Authentication

WAMP Challenge-Response ("WAMP-CRA") authentication is a simple, secure authentication mechanism using a shared secret. The client and the server share a secret. The secret never travels the wire, hence WAMP-CRA can be used via non-TLS connections.

wampy needs the secret to be set as an environment variable against the key WAMPYSECRET on deployment or in the test environment (if testing auth) otherwise a WampyError will be raised. In future a Client could take configuration on startup.

The Router must also be configured to expect Users and by what auth method.

For the Client you can instantiate the Client with roles which can take authmethods and authid.

```
roles = {
    'roles': {
        'subscriber': {},
        'publisher': {},
        'callee': {
            'shared_registration': True,
        },
        'caller': {},
    },
    'authmethods': ['wampcra']  # where "anonymous" is the default
    'authid': 'your-username-or-identifier'
}

client = Client(roles=roles)
```

And the Router would include something like...

```
"auth": {
    "wampcra": {
        "type": "static",
        "role": "wampy",
        "users": {
            "your-username-or-identifier": {
                "secret": "prq7+YkJ1/KlW1X0YczMHw==",
                "role": "wampy",
                "salt": "salt123",
                "iterations": 100,
                "keylen": 16,
            },
            "someone-else": {
                "secret": "secret2",
                "role": "wampy",
                ...
            },
            ...
        }
    },
    "anonymous": {
        "type": "static",
        "role": "wampy"
    }
}
```

with permissions for RPC and subscriptions optionally defined. See the included testing config for a more complete example.

The MessageHandler Class

Every wampy `Client` requires a `MessageHandler`. This is a class with a list of `Messages` it will accept and a "handle" method for each.

The default `MessageHandler` contains everything you need to use WAMP in your microservices, but you may want to add more behaviour such as logging messages, encrypting messages, appending meta data or custom authorisation.

If you want to define your own `MessageHandler` then you must subclass the default and override the "handle" methods for each `Message` customisation you need.

Note that whenever the `Session` receives a `Message` it calls `handle_message` on the `MessageHandler`. You can override this if you want to add global behaviour changes. `handle_message` will delegate to specific handlers, e.g. `handle_invocation`.

For example.

```
from wampy.message_handler import MessageHandler


class CustomHandler(MessageHandler):

    def handle_welcome(self, message_obj):
        # maybe do some auth stuff here
        super(CustomHandler, self).handle_welcome(message_obj)
        # and maybe then some other stuff now like alerting
```

There may be no need to even do what wampy does if your application already has patterns for handling WAMP messages! In which case override but don't call `super` - just do your own thing.

Then your Client should be initialised with an *instance* of the custom handler.

```
from wampy.peers.clients import Client

client = Client(message_handler=CustomHandler())
```

Testing wampy apps

To test any WAMP application you are going to need a Peer acting as a Router.

wampy provides a `pytest` fixture for this: `router` which must be installed via `$ pip install --editable .[dev]`. Usage is then simple.

For example

```
def test_my_wampy_applications(router):
    # do stuff here
```

The router is **Crossbar.io** and will be started and shutdown between each test.

It has a default configuration which you can override in your tests by creating a `config_path` fixture in your own `conftest` or test module.

For example

```
import pytest
```

```
@pytest.fixture
def config_path():
    return './path/to/my/preferred/crossbar.json'
```

Now any test using `router` will be a Crossbar.io server configured yourself.

For example

```
def test_my_app(router):
    # this router's configuration has been overridden!
```

If you require even more control you can import the router itself from `wampy.peers.routers` and setup your tests however you need to.

wampy also provides a `pytest` fixture for a WAMP client.

Here is an example testing a wampy `HelloService` application.

```
import pytest

from wampy.roles.callee import callee
from wampy.peers.clients import Client
from wampy.testing import wait_for_registrations

class HelloService(Client):

    @callee
    def say_hello(self, name):
        message = "Hello {}".format(name)
        return message


@pytest.yield_fixture
def hello_service(router):
    with HelloService(router=router) as service:
        wait_for_registrations(service, 1)
        yield


def test_get_hello_message(hello_service, router, client):
    response = client.rpc.say_hello(name="wampy")

    assert response == "Hello wampy"
```

Notice the use of `wait_for_registrations`. All wampy actions are asynchronous, and so it's easy to get confused when setting up tests wondering why an application hasn't registered Callees or subscribed to Topics, or a Session even opened yet.

So to help you setup your tests and avoid race conditions there are some helpers that you can execute to wait for async certain actions to perform before you start actually running test cases. These are:

```
# execute with the client you're waiting for as the only argument
from wampy.testing import wait_for_session
# e.g. ```wait_for_session(client)```

# wait for a specific number of registrations on a client
from wampy.testing import wait_for_registrations
# e.g. ``wait_for_registrations(client, number_of_registrations=5)
```

```
# wait for a specific number of subscriptions on a client
from wampy.testing import wait_for_subscriptions
# e.g. ``wait_for_subscriptions(client, number_of_subscriptions=7)

# provied a function that raises until the test passes
from test.helpers import assert_stops_raising
# e.g. assert_stops_raising(my_func_that_raises_until_condition_met)
```

For far more examples, see the wampy test suite.

TLS/wss Support

Your Router must be configured to use TLS. For an example see the config used by the test runner along with the TLS Router setup.

To connect a Client over TLS you must provide a connection URL using the `wss` protocol and your **Router** probably will require you to provide a certificate for authorisation.

```
In [1]: from wampy.peers import Client

In [2]: client = Client(url="wss://...", cert_path="./...")
```

CHAPTER 3

modules

wampy.constants module

wampy.errors module

exception `errors.`**`ConnectionError`**
    Bases: `exceptions.Exception`

exception `errors.`**`IncompleteFrameError`**`(`*required_bytes*`)`
    Bases: `exceptions.Exception`

exception `errors.`**`WampProtocolError`**
    Bases: `exceptions.Exception`

exception `errors.`**`WampyError`**
    Bases: `exceptions.Exception`

exception `errors.`**`WebsocktProtocolError`**
    Bases: `exceptions.Exception`

wampy.roles.callee module

class `mixins.`**`ParseUrlMixin`**
    Bases: `object`

`parse_url``()`
    Parses a URL of the form:

- ws://host[:port][path]
- wss://host[:port][path]
- ws+unix:///path/to/my.socket

wampy.session module

class `session.`**`Session`**`(`*client*, *router*, *transport*, *message_handler*`)`
    Bases: `object`

A transient conversation between two Peers attached to a Realm and running over a Transport.

WAMP Sessions are established over a WAMP Connection which is the responsibility of the `Transport` object.

Each wampy `Session` manages its own WAMP connection via the `Transport`.

Once the connection is established, the Session is begun when the Realm is joined. This is achieved by sending the HELLO message.

Note: Routing occurs only between WAMP Sessions that have joined the same Realm.

begin()

end()

host

id

port

realm

recv_message(*timeout=5*)

roles

send_message(*message_obj*)

wampy.messages.call module

class `call.`**Call**(*procedure*, *options=None*, *args=None*, *kwargs=None*)

Bases: `object`

When a Caller wishes to call a remote procedure, it sends a “CALL” message to a Dealer.

Message is of the format `[CALL,Request|id,Options|dict,Procedure|uri,Arguments|list,ArgumentsKw`
e.g.

```
[
    CALL, 10001, {}, "com.myapp.myprocedure1", [], {}
]
```

“Request” is a random, ephemeral ID chosen by the Callee and used to correlate the Dealer’s response with the request.

“Options” is a dictionary that allows to provide additional registration request details in a extensible way.

WAMP_CODE = 48

message

name = ‘call’

wampy.messages.hello module

class `hello.`**Hello**(*realm*, *roles*)

Bases: `object`

Send a HELLO message to the Router.

Message is of the format `[HELLO, Realm|uri, Details|dict]`, e.g.

```
[
    HELLO, "realm", {
        "roles": {"subscriber": {}, "publisher": {}},
        "authmethods": ["wampcra"],
        "authid": "peter"
    }
]
```

WAMP_CODE = 1

message

name = 'hello'

wampy.messages.goodbye module

class `goodbye.`**Goodbye**(*details=None*, *reason='wamp.close.normal'*)

Bases: `object`

Send a GOODBYE message to the Router.

Message is of the format `[GOODBYE, Details|dict, Reason|uri]`, e.g.

```
[
    GOODBYE, {}, "wamp.close.normal"
]
```

DEFAULT_REASON = 'wamp.close.normal'

WAMP_CODE = 6

message

name = 'goodbye'

wampy.messages.subscribe module

class `subscribe.`**Subscribe**(*topic*, *options=None*)

Bases: `object`

Send a SUBSCRIBE message to the Router.

Message is of the format `[SUBSCRIBE, Request|id, Options|dict, Topic|uri]`, e.g.

```
[
    32, 713845233, {}, "com.myapp.mytopic1"
]
```

WAMP_CODE = 32

message

name = 'subscribe'

wampy.messages.publish module

class `publish.`**`Publish`**(*topic*, *options*, **args*, ***kwargs*)

Bases: `object`

Send a PUBLISH message to the Router.

Message is of the format `[PUBLISH,Request|id,Options|dict,Topic|uri,Arguments|list,ArgumentsKw|`
e.g.

```
[
    16, 239714735, {}, "com.myapp.mytopic1", [],
    {"color": "orange", "sizes": [23, 42, 7]}
]
```

`WAMP_CODE` **= 16**

`message`

`name` **= 'publish'**

wampy.messages.yield module

class `yield_.`**`Yield`**(*invocation_request_id*, *options=None*, *result_args=None*, *result_kwargs=None*)

Bases: `object`

When the Callee is able to successfully process and finish the execution of the call, it answers by sending a "YIELD" message to the Dealer.

Message is of the format

```
[
    YIELD, INVOCATION.Request|id, Options|dict, Arguments|list,
    ArgumentsKw|dict
]
```

"INVOCATION.Request" is the ID from the original invocation request.

"Options"is a dictionary that allows to provide additional options.

"Arguments" is a list of positional result elements (each of arbitrary type). The list may be of zero length.

"ArgumentsKw" is a dictionary of keyword result elements (each of arbitrary type). The dictionary may be empty.

`WAMP_CODE` **= 70**

`message`

`name` **= 'yield'**

wampy.messages.register module

class `register.`**`Register`**(*procedure*, *options=None*)

Bases: `object`

A Callee announces the availability of an endpoint implementing a procedure with a Dealer by sending a "REGISTER" message.

Message is of the format `[REGISTER, Request|id, Options|dict, Procedure|uri]`, e.g.

```
[
    REGISTER, 25349185, {}, "com.myapp.myprocedure1"
]
```

"Request" is a random, ephemeral ID chosen by the Callee and used to correlate the Dealer's response with the request.

"Options" is a dictionary that allows to provide additional registration request details in a extensible way.

`WAMP_CODE` = **64**

`message`

`name` = **'register'**

wampy.peers.clients module

class `clients.`**`Client`**(*url=None, cert_path=None, realm='realm1', roles={'authmethods': ['anonymous'], 'roles': {'subscriber': {}, 'publisher': {}, 'caller': {}, 'callee': {'shared_registration': True}}}, message_handler=None, name=None, router=None*)

Bases: `object`

A WAMP Client for use in Python applications, scripts and shells.

`call`

`make_rpc`(*message*)

`publish`

`recv_message`()

`register_roles`()

`registration_map`

`request_ids`

`rpc`

`send_message`(*message*)

`session`

`start`()

`stop`()

`subscription_map`

wampy.peers.routers module

class `routers.`**`Crossbar`**(*url='ws://localhost:8080', config_path='./crossbar/config.json', crossbar_directory=None*)

Bases: `wampy.mixins.ParseUrlMixin`

`can_use_tls`

start()
Start Crossbar.io in a subprocess.

stop()

try_connection()

class routers.**Router**(*url*, *cert_path=None*, *ipv=4*)
Bases: `wampy.mixins.ParseUrlMixin`

wampy.roles.callee module

class callee.**RegisterProcedureDecorator**(**args*, ***kwargs*)
Bases: `object`

classmethod decorator(**args*, ***kwargs*)

wampy.roles.caller module

class caller.**CallProxy**(*client*)
Proxy wrapper of a *wampy* client for WAMP application RPCs.

Applictions and their endpoints are identified by dot delimented strings, e.g.

```
"com.example.endpoints"
```

and a *CallProxy* object will call such and endpoint, passing in any *args* or *kwargs* necessary.

class caller.**RpcProxy**(*client*)
Proxy wrapper of a *wampy* client for WAMP application RPCs where the endpoint is a non-delimted single string name, such as a function name, e.g.

```
"get_data"
```

The typical use case of this proxy class is for microservices where endpoints are class methods.

wampy.roles.publisher module

class publisher.**PublishProxy**(*client*)

wampy.roles.subscriber module

class subscriber.**RegisterSubscriptionDecorator**(***kwargs*)
Bases: `object`

subscriber.**subscribe**
alias of *RegisterSubscriptionDecorator*

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index