tinyrpc Documentation

Release 0.6dev

Marc Brinkmann

Aug 14, 2017

Contents

1 Quickstart examples **3**
- 1.1 HTTP based . . . 3
- 1.2 0mq . . . 4

2 Further examples **7**

3 Table of contents **9**
- 3.1 Structure of tinyrpc . . . 9
- 3.2 The protocol layer . . . 9
- 3.3 Dispatching . . . 14
- 3.4 Transports . . . 17
- 3.5 RPC Client . . . 20
- 3.6 Server implementations . . . 21
- 3.7 Exception reference . . . 22

4 People **23**
- 4.1 Creator . . . 23
- 4.2 Maintainer . . . 23
- 4.3 Contributors . . . 23

Python Module Index **25**

`tinyrpc` is a library for making and handling RPC calls in python. Its initial scope is handling jsonrpc, although it aims to be very well-documented and modular to make it easy to add support for further protocols.

A feature is support of multiple transports (or none at all) and providing clever syntactic sugar for writing dispatchers.

CHAPTER 1

Quickstart examples

The source contains all of these examples in a working fashion in the examples subfolder.

HTTP based

A client making JSONRPC calls via HTTP (this requires `requests` to be installed):

```
from tinyrpc.protocols.jsonrpc import JSONRPCProtocol
from tinyrpc.transports.http import HttpPostClientTransport
from tinyrpc import RPCClient

rpc_client = RPCClient(
    JSONRPCProtocol(),
    HttpPostClientTransport('http://example.org/jsonrpc/2.0/')
)

time_server = rpc_client.get_proxy()

# ...

# call a method called 'get_time_in' with a single string argument
time_in_berlin = time_server.get_time_in('Europe/Berlin')
```

These can be answered by a server implemented as follows:

```
import gevent
import gevent.wsgi
import gevent.queue
from tinyrpc.protocols.jsonrpc import JSONRPCProtocol
from tinyrpc.transports.wsgi import WsgiServerTransport
from tinyrpc.server.gevent import RPCServerGreenlets
from tinyrpc.dispatch import RPCDispatcher

dispatcher = RPCDispatcher()
```

```
transport = WsgiServerTransport(queue_class=gevent.queue.Queue)

# start wsgi server as a background-greenlet
wsgi_server = gevent.wsgi.WSGIServer(('127.0.0.1', 80), transport.handle)
gevent.spawn(wsgi_server.serve_forever)

rpc_server = RPCServerGreenlets(
    transport,
    JSONRPCProtocol(),
    dispatcher
)

@dispatcher.public
def reverse_string(s):
    return s[::-1]

# in the main greenlet, run our rpc_server
rpc_server.serve_forever()
```

0mq

An example using `zmq` is very similiar, differing only in the instantiation of the transport:

```
import zmq

from tinyrpc.protocols.jsonrpc import JSONRPCProtocol
from tinyrpc.transports.zmq import ZmqClientTransport
from tinyrpc import RPCClient

ctx = zmq.Context()

rpc_client = RPCClient(
    JSONRPCProtocol(),
    ZmqClientTransport.create(ctx, 'tcp://127.0.0.1:5001')
)

remote_server = rpc_client.get_proxy()

# call a method called 'reverse_string' with a single string argument
result = remote_server.reverse_string('Hello, World!')

print "Server answered:", result
```

Matching server:

```
import zmq

from tinyrpc.protocols.jsonrpc import JSONRPCProtocol
from tinyrpc.transports.zmq import ZmqServerTransport
from tinyrpc.server import RPCServer
from tinyrpc.dispatch import RPCDispatcher

ctx = zmq.Context()
dispatcher = RPCDispatcher()
transport = ZmqServerTransport.create(ctx, 'tcp://127.0.0.1:5001')
```

```
rpc_server = RPCServer(
    transport,
    JSONRPCProtocol(),
    dispatcher
)

@dispatcher.public
def reverse_string(s):
    return s[::-1]

rpc_server.serve_forever()
```

CHAPTER 2

Further examples

In *The protocol layer*, you can find client and server examples on how to use just the protocol parsing parts of `tinyrpc`.

The *`RPCDispatcher`* should be useful on its own (or at least easily replaced with one of your choosing), see *Dispatching* for details.

CHAPTER 3

Table of contents

Structure of tinyrpc

`tinyrpc` architectually considers three layers: Transport, Protocol and Dispatch.

The Transport-layer is responsible for receiving and sending messages. No assumptions are made about messages, except that they are of a fixed size. Messages are received and possibly passed on a Python strings.

In an RPC context, messages coming in (containing requests) are simply called messages, a message sent in reply is called a reply. Replies are always serialized responses.

On the Protocol-layer messages are decoded into a format that is protocol independent, i.e. incoming messages are turned into requests or vice verse, while outgoing messages can be turned from responses into replies or the other way around.

The Dispatch-layer performs the actual method calling and serializes the return value. These can be routed back through the Protocol- and Transport-layer to return the answer to the calling client.

Each layer is useful "on its own" and can be used seperately. If you simply need to decode a jsonrpc message, without passing it on or sending it through a transport, any *`RPCProtocol`*-class is completely usable on its own.

The protocol layer

Any protocol is implemented by deriving from *`RPCProtocol`* and implementing all of its members:

class `tinyrpc.`**`RPCProtocol`**
Base class for all protocol implementations.

`create_request`(*method*, *args=None*, *kwargs=None*, *one_way=False*)
Creates a new RPCRequest object.

It is up to the implementing protocol whether or not `args`, `kwargs`, one of these, both at once or none of them are supported.

Parameters

- **method** – The method name to invoke.
- **args** – The positional arguments to call the method with.
- **kwargs** – The keyword arguments to call the method with.
- **one_way** – The request is an update, i.e. it does not expect a reply.

Returns A new *RPCRequest* instance.

parse_reply(*data*)
Parses a reply and returns an `RPCResponse` instance.

Returns An instanced response.

parse_request(*data*)
Parses a request given as a string and returns an `RPCRequest` instance.

Returns An instanced request.

These require implementations of the following classes as well:

class `tinyrpc.`**RPCRequest**

error_respond(*error*)
Creates an error response.

Create a response indicating that the request was parsed correctly, but an error has occured trying to fulfill it.

Parameters **error** – An exception or a string describing the error.

Returns A response or `None` to indicate that no error should be sent out.

respond(*result*)
Create a response.

Call this to return the result of a successful method invocation.

This creates and returns an instance of a protocol-specific subclass of *RPCResponse*.

Parameters **result** – Passed on to new response instance.

Returns A response or `None` to indicate this request does not expect a response.

serialize()
Returns a serialization of the request.

Returns A string to be passed on to a transport.

class `tinyrpc.`**RPCResponse**
RPC call response class.

Base class for all deriving responses.

Has an attribute `result` containing the result of the RPC call, unless an error occured, in which case an attribute `error` will contain the error message.

serialize()
Returns a serialization of the response.

Returns A reply to be passed on to a transport.

class `tinyrpc.`**BadRequestError**
Base class for all errors that caused the processing of a request to abort before a request object could be instantiated.

error_respond()
Create RPCErrorResponse to respond the error.

Returns A error responce instance or None, if the protocol decides to drop the error silently.

Every protocol deals with multiple kinds of structures: data arguments are always byte strings, either messages or replies, that are sent via or received from a transport.

There are two protocol-specific subclasses of *RPCRequest* and *RPCResponse*, these represent well-formed requests and responses.

Finally, if an error occurs during parsing of a request, a *BadRequestError* instance must be thrown. These need to be subclassed for each protocol as well, since they generate error replies.

Batch protocols

Some protocols may support batch requests. In this case, they need to derive from *RPCBatchProtocol*.

Batch protocols differ in that their *parse_request()* method may return an instance of *RPCBatchRequest*. They also possess an addional method in *create_batch_request()*.

Handling a batch request is slightly different, while it supports error_respond(), to make actual responses, *create_batch_response()* needs to be used.

No assumptions are made whether or not it is okay for batch requests to be handled in parallel. This is up to the server/dispatch implementation, which must be chosen appropriately.

class tinyrpc.**RPCBatchProtocol**

create_batch_request(*requests=None*)
Create a new *tinyrpc.RPCBatchRequest* object.

Parameters **requests** – A list of requests.

class tinyrpc.**RPCBatchRequest**
Multiple requests batched together.

A batch request is a subclass of list. Protocols that support multiple requests in a single message use this to group them together.

Handling a batch requests is done in any order, responses must be gathered in a batch response and be in the same order as their respective requests.

Any item of a batch request is either a request or a subclass of *BadRequestError*, which indicates that there has been an error in parsing the request.

create_batch_response()
Creates a response suitable for responding to this request.

Returns An *RPCBatchResponse* or None, if no response is expected.

class tinyrpc.**RPCBatchResponse**
Multiple response from a batch request. See *RPCBatchRequest* on how to handle.

Items in a batch response need to be *RPCResponse* instances or None, meaning no reply should generated for the request.

serialize()
Returns a serialization of the batch response.

Supported protocols

Any supported protocol is used by instantiating its class and calling the interface of *RPCProtocol*. Note that constructors are not part of the interface, any protocol may have specific arguments for its instances.

Protocols usually live in their own module because they may need to import optional modules that needn't be a dependency for all of `tinyrpc`.

Example

The following example shows how to use the `JSONRPCProtocol` class in a custom application, without using any other components:

Server

```
from tinyrpc.protocols.jsonrpc import JSONRPCProtocol
from tinyrpc import BadRequestError, RPCBatchRequest

rpc = JSONRPCProtocol()

# the code below is valid for all protocols, not just JSONRPC:

def handle_incoming_message(self, data):
    try:
        request = rpc.parse_request(data)
    except BadRequestError as e:
        # request was invalid, directly create response
        response = e.error_respond(e)
    else:
        # we got a valid request
        # the handle_request function is user-defined
        # and returns some form of response
        if hasattr(request, create_batch_response):
            response = request.create_batch_response(
                handle_request(req) for req in request
            )
        else:
            response = handle_request(request)

    # now send the response to the client
    if response != None:
         send_to_client(response.serialize())


def handle_request(request):
    try:
        # do magic with method, args, kwargs...
        return request.respond(result)
    except Exception as e:
        # for example, a method wasn't found
        return request.error_respond(e)
```

Client

```
from tinyrpc.protocols.jsonrpc import JSONRPCProtocol

rpc = JSONRPCProtocol()

# again, code below is protocol-independent

# assuming you want to call method(*args, **kwargs)

request = rpc.create_request(method, args, kwargs)
reply = send_to_server_and_get_reply(request)

response = rpc.parse_reply(reply)

if hasattr(response, 'error'):
    # error handling...
else:
    # the return value is found in response.result
    do_something_with(response.result)
```

Another example, this time using batch requests:

```
# or using batch requests:

requests = rpc.create_batch_request([
    rpc.create_request(method_1, args_1, kwargs_1)
    rpc.create_request(method_2, args_2, kwargs_2)
    # ...
])

reply = send_to_server_and_get_reply(request)

responses = rpc.parse_reply(reply)

for responses in response:
    if hasattr(reponse, 'error'):
        # ...
```

Finally, one-way requests are requests where the client does not expect an answer:

```
request = rpc.create_request(method, args, kwargs, one_way=True)
send_to_server(request)

# done
```

JSON-RPC

class tinyrpc.protocols.jsonrpc.**JSONRPCProtocol**(**args*, ***kwargs*)

JSONRPC protocol implementation.

Currently, only version 2.0 is supported.

Dispatching

Dispatching in `tinyrpc` is very similiar to url-routing in web frameworks. Functions are registered with a specific name and made public, i.e. callable, to remote clients.

Examples

Exposing a few functions:

```
from tinyrpc.dispatch import RPCDispatcher

dispatch = RPCDispatcher()

@dispatch.public
def foo():
    # ...

@dispatch.public
def bar(arg):
    # ...

# later on, assuming we know we want to call foo(*args, **kwargs):

f = dispatch.get_method('foo')
f(*args, **kwargs)
```

Using prefixes and instance registration:

```
from tinyrpc.dispatch import public

class SomeWebsite(object):
    def __init__(self, ...):
        # note: this method will not be exposed

    def secret(self):
        # another unexposed method

    @public
    def get_user_info(self, user):
        # ...

    # using a different name
    @public('get_user_comment')
    def get_comment(self, comment_id):
        # ...
```

The code above declares an RPC interface for `SomeWebsite` objects, consisting of two visible methods: `get_user_info(user)` and `get_user_comment(commend_id)`.

These can be used with a dispatcher now:

```
def hello():
    # ...
```

```
website1 = SomeWebsite(...)
website2 = SomeWebsite(...)

from tinyrpc.dispatch import RPCDispatcher

dispatcher = RPCDispatcher()

# directly register version method
@dispatcher.public
def version():
    # ...

# add earlier defined method
dispatcher.add_method(hello)

# register the two website instances
dispatcher.register_instance(website1, 'sitea.')
dispatcher.register_instance(website2, 'siteb.')
```

In the example above, the *RPCDispatcher* now knows a total of six registered methods: `version`, `hello`, `sitea.get_user_info`, `sitea.get_user_comment`, `siteb.get_user_info`, `siteb.get_user_comment`.

Automatic dispatching

When writing a server application, a higher level dispatching method is available with *dispatch()*:

```
from tinyrpc.dispatch import RPCDispatcher

dispatcher = RPCDispatcher()

# register methods like in the examples above
# ...
# now assumes that a valid RPCRequest has been obtained, as `request`

response = dispatcher.dispatch(request)

# response can be directly processed back to the client, all Exceptions have
# been handled already
```

API reference

class `tinyrpc.dispatch.`**RPCDispatcher**

Stores name-to-method mappings.

add_method(*f*, *name=None*)

Add a method to the dispatcher.

Parameters

- **f** – Callable to be added.
- **name** – Name to register it with. If `None`, `f.__name__` will be used.

add_subdispatch(*dispatcher*, *prefix=''*)

Adds a subdispatcher, possibly in its own namespace.

Parameters

- **dispatcher** – The dispatcher to add as a subdispatcher.
- **prefix** – A prefix. All of the new subdispatchers methods will be available as prefix + their original name.

dispatch(*request*, *caller=None*)
Fully handle request.

The dispatch method determines which method to call, calls it and returns a response containing a result.

No exceptions will be thrown, rather, every exception will be turned into a response using *error_respond()*.

If a method isn't found, a *MethodNotFoundError* response will be returned. If any error occurs outside of the requested method, a *ServerError* without any error information will be returend.

If the method is found and called but throws an exception, the exception thrown is used as a response instead. This is the only case in which information from the exception is possibly propagated back to the client, as the exception is part of the requested method.

RPCBatchRequest instances are handled by handling all its children in order and collecting the results, then returning an *RPCBatchResponse* with the results.

To allow for custom processing around calling the method (i.e. custom error handling), the optional parameter `caller` may be provided with a callable. When present invoking the method is deferred to this callable.

Parameters

- **request** – An *RPCRequest()*.
- **caller** – An optional callable used to invoke the method.

Returns An *RPCResponse()*.

get_method(*name*)
Retrieve a previously registered method.

Checks if a method matching `name` has been registered.

If `get_method()` cannot find a method, every subdispatcher with a prefix matching the method name is checked as well.

If a method isn't found, a `KeyError` is thrown.

Parameters

- **name** – Callable to find.
- **return** – The callable.

public(*name=None*)
Convenient decorator.

Allows easy registering of functions to this dispatcher. Example:

```
dispatch = RPCDispatcher()

@dispatch.public
def foo(bar):
    # ...

class Baz(object):
    def not_exposed(self):
```

```
    # ...

@dispatch.public(name='do_something')
def visible_method(arg1)
    # ...
```

Parameters **name** – Name to register callable with

register_instance(*obj*, *prefix=''*)
Create new subdispatcher and register all public object methods on it.

To be used in conjunction with the *tinyrpc.dispatch.public()* decorator (*not* *tinyrpc.dispatch.RPCDispatcher.public()*).

Parameters

- **obj** – The object whose public methods should be made available.
- **prefix** – A prefix for the new subdispatcher.

Classes can be made to support an RPC interface without coupling it to a dispatcher using a decorator:

tinyrpc.dispatch.**public**(*name=None*)
Set RPC name on function.

This function decorator will set the `_rpc_public_name` attribute on a function, causing it to be picked up if an instance of its parent class is registered using *register_instance()*.

`@public` is a shortcut for `@public()`.

Parameters **name** – The name to register the function with.

Transports

Transports are somewhat low level interface concerned with transporting messages across through different means. "Messages" in this case are simple strings. All transports need to support two different interfaces:

class tinyrpc.transports.**ServerTransport**
Base class for all server transports.

receive_message()
Receive a message from the transport.

Blocks until another message has been received. May return a context opaque to clients that should be passed on `send_reply()` to identify the client later on.

Returns A tuple consisting of `(context, message)`.

send_reply(*context*, *reply*)
Sends a reply to a client.

The client is usually identified by passing `context` as returned from the original `receive_message()` call.

Messages must be strings, it is up to the sender to convert the beforehand. A non-string value raises a `TypeError`.

Parameters

- **context** – A context returned by `receive_message()`.

- **reply** – A string to send back as the reply.

class tinyrpc.transports.**ClientTransport**
Base class for all client transports.

send_message(*message*, *expect_reply=True*)
Send a message to the server and possibly receive a reply.

Sends a message to the connected server.

Messages must be strings, it is up to the sender to convert the beforehand. A non-string value raises a TypeError.

This function will block until one reply has been received.

Parameters **message** – A string to send.

Returns A string containing the server reply.

Note that these transports are of relevance when using tinyrpc-built in facilities. They can be coopted for any other purpose, if you simply need reliable server-client message passing as well.

Transport implementations

A few transport implementations are included with tinyrpc:

0mq

Based on zmq, supports 0mq based sockets. Highly recommended:

class tinyrpc.transports.zmq.**ZmqServerTransport**(*socket*)
Server transport based on a zmq.ROUTER socket.

Parameters **socket** – A zmq.ROUTER socket instance, bound to an endpoint.

classmethod **create**(*zmq_context*, *endpoint*)
Create new server transport.

Instead of creating the socket yourself, you can call this function and merely pass the zmq.core.context.Context instance.

By passing a context imported from zmq.green, you can use green (gevent) 0mq sockets as well.

Parameters

- **zmq_context** – A 0mq context.
- **endpoint** – The endpoint clients will connect to.

class tinyrpc.transports.zmq.**ZmqClientTransport**(*socket*)
Client transport based on a zmq.REQ socket.

Parameters **socket** – A zmq.REQ socket instance, connected to the server socket.

classmethod **create**(*zmq_context*, *endpoint*)
Create new client transport.

Instead of creating the socket yourself, you can call this function and merely pass the zmq.core.context.Context instance.

By passing a context imported from zmq.green, you can use green (gevent) 0mq sockets as well.

Parameters

- **zmq_context** – A 0mq context.
- **endpoint** – The endpoint the server is bound to.

HTTP

There is only an HTTP client, no server (use WSGI instead).

WSGI

class tinyrpc.transports.wsgi.**WsgiServerTransport**(*max_content_length=4096*, *queue_class=<class Queue.Queue>*, *allow_origin='*'*)

WSGI transport.

Requires werkzeug.

Due to the nature of WSGI, this transport has a few pecularities: It must be run in a thread, greenlet or some other form of concurrent execution primitive.

This is due to *handle()* blocking while waiting for a call to send_reply().

The parameter queue_class must be used to supply a proper queue class for the chosen concurrency mechanism (i.e. when using gevent, set it to gevent.queue.Queue).

Parameters

- **max_content_length** – The maximum request content size allowed. Should be set to a sane value to prevent DoS-Attacks.
- **queue_class** – The Queue class to use.
- **allow_origin** – The Access-Control-Allow-Origin header. Defaults to * (so change it if you need actual security).

handle(*environ*, *start_response*)

WSGI handler function.

The transport will serve a request by reading the message and putting it into an internal buffer. It will then block until another concurrently running function sends a reply using send_reply().

The reply will then be sent to the client being handled and handle will return.

CGI

class tinyrpc.transports.cgi.**CGIServerTransport**

CGI transport.

Reading stdin is blocking but, given that we've been called, something is waiting. The transport accepts both GET and POST request.

A POST request provides the entire JSON-RPC request in the body of the HTTP request.

A GET request provides the elements of the JSON-RPC request in separate query parameters and only the params field contains a JSON object or array. i.e. curl 'http://server?jsonrpc=2.0&id=1&method="doit"&params={"arg"="something"}'

receive_message()

Receive a message from the transport.

Blocks until another message has been received. May return a context opaque to clients that should be passed on `send_reply()` to identify the client later on.

Returns A tuple consisting of `(context, message)`.

send_reply(*context*, *reply*)
Sends a reply to a client.

The client is usually identified by passing `context` as returned from the original `receive_message()` call.

Messages must be strings, it is up to the sender to convert the beforehand. A non-string value raises a `TypeError`.

Parameters

- **context** – A context returned by `receive_message()`.
- **reply** – A string to send back as the reply.

Callback

class tinyrpc.transports.callback.**CallbackServerTransport**(*reader*, *writer*)
Callback server transport.

Used when tinyrpc is part of a system where it cannot directly attach to a socket or stream. The methods `receive_message()` and `send_reply()` are implemented by callback functions that were passed to `__init__()`.

This transport is also useful for testing the other modules of tinyrpc.

receive_message()
Receive a message from the transport.

Uses the callback function `reader` to obtain a json string. May return a context opaque to clients that should be passed on `send_reply()` to identify the client later on.

Returns A tuple consisting of `(context, message)`.

send_reply(*context*, *reply*)
Sends a reply to a client.

The client is usually identified by passing `context` as returned from the original `receive_message()` call.

Messages must be a string, it is up to the sender to convert it beforehand. A non-string value raises a `TypeError`.

Parameters

- **context** – A context returned by `receive_message()`.
- **reply** – A string to send back as the reply.

RPC Client

RPCClient instances are high-level handlers for making remote procedure calls to servers. Other than *RPCProxy* objects, they are what most user applications interact with.

Clients needs to be instantiated with a protocol and a transport to function. Proxies are syntactic sugar for using clients.

class tinyrpc.client.**RPCClient**(*protocol*, *transport*)
Client for making RPC calls to connected servers.

Parameters

- **protocol** – An *RPCProtocol* instance.
- **transport** – A *ClientTransport* instance.

batch_call(*calls*)
Experimental, use at your own peril.

call(*method*, *args*, *kwargs*, *one_way=False*)
Calls the requested method and returns the result.

If an error occured, an *RPCError* instance is raised.

Parameters

- **method** – Name of the method to call.
- **args** – Arguments to pass to the method.
- **kwargs** – Keyword arguments to pass to the method.
- **one_way** – Whether or not a reply is desired.

get_proxy(*prefix=''*, *one_way=False*)
Convenience method for creating a proxy.

Parameters

- **prefix** – Passed on to *RPCProxy*.
- **one_way** – Passed on to *RPCProxy*.

Returns *RPCProxy* instance.

class tinyrpc.client.**RPCProxy**(*client*, *prefix=''*, *one_way=False*)
Create a new remote proxy object.

Proxies allow calling of methods through a simpler interface. See the documentation for an example.

Parameters

- **client** – An *RPCClient* instance.
- **prefix** – Prefix to prepend to every method name.
- **one_way** – Passed to every call of `call()`.

Server implementations

Like *RPC Client*, servers are top-level instances that most user code should interact with. They provide runnable functions that are combined with transports, protocols and dispatchers to form a complete RPC system.

class tinyrpc.server.**RPCServer**(*transport*, *protocol*, *dispatcher*)
High level RPC server.

Parameters

- **transport** – The `RPCTransport` to use.
- **protocol** – The *RPCProtocol* to use.
- **dispatcher** – The *RPCDispatcher* to use.

receive_one_message()
Handle a single request.

Polls the transport for a new message.

After a new message has arrived _spawn() is called with a handler function and arguments to handle the request.

The handler function will try to decode the message using the supplied protocol, if that fails, an error response will be sent. After decoding the message, the dispatcher will be asked to handle the resultung request and the return value (either an error or a result) will be sent back to the client using the transport.

serve_forever()
Handle requests forever.

Starts the server loop continuously calling *receive_one_message()* to process the next incoming request.

class tinyrpc.server.gevent.**RPCServerGreenlets**
Asynchronous RPCServer.

This implementation of *RPCServer* uses gevent.spawn() to spawn new client handlers, result in asynchronous handling of clients using greenlets.

Exception reference

exception tinyrpc.exc.**BadReplyError**
Base class for all errors that caused processing of a reply to abort before it could be turned in a response object.

exception tinyrpc.exc.**BadRequestError**
Base class for all errors that caused the processing of a request to abort before a request object could be instantiated.

error_respond()
Create RPCErrorResponse to respond the error.

Returns A error responce instance or None, if the protocol decides to drop the error silently.

exception tinyrpc.exc.**InvalidReplyError**
A reply received was malformed (i.e. violated the specification) and could not be parsed into a response.

exception tinyrpc.exc.**InvalidRequestError**
A request made was malformed (i.e. violated the specification) and could not be parsed.

exception tinyrpc.exc.**MethodNotFoundError**
The desired method was not found.

exception tinyrpc.exc.**RPCError**
Base class for all excetions thrown by tinyrpc.

exception tinyrpc.exc.**ServerError**
An internal error in the RPC system occured.

CHAPTER 4

People

Creator

- Marc Brinkmann: https://github.com/mbr

Maintainer

- Leo Noordergraaf: https://github.com/lnoor

Contributors

- Guilherme Salgado: https://github.com/gsalgado
- jnnk: https://github.com/jnnk
- Satoshi Kobayashi: https://github.com/satosi-k

Python Module Index

t

tinyrpc.exc, 22
tinyrpc.server, 21

Index

A

add_method() (tinyrpc.dispatch.RPCDispatcher method), 15
add_subdispatch() (tinyrpc.dispatch.RPCDispatcher method), 15

B

BadReplyError, 22
BadRequestError, 22
BadRequestError (class in tinyrpc), 10
batch_call() (tinyrpc.client.RPCClient method), 21

C

call() (tinyrpc.client.RPCClient method), 21
CallbackServerTransport (class in tinyrpc.transports.callback), 20
CGIServerTransport (class in tinyrpc.transports.cgi), 19
ClientTransport (class in tinyrpc.transports), 18
create() (tinyrpc.transports.zmq.ZmqClientTransport class method), 18
create() (tinyrpc.transports.zmq.ZmqServerTransport class method), 18
create_batch_request() (tinyrpc.RPCBatchProtocol method), 11
create_batch_response() (tinyrpc.RPCBatchRequest method), 11
create_request() (tinyrpc.RPCProtocol method), 9

D

dispatch() (tinyrpc.dispatch.RPCDispatcher method), 16

E

error_respond() (tinyrpc.BadRequestError method), 10
error_respond() (tinyrpc.exc.BadRequestError method), 22
error_respond() (tinyrpc.RPCRequest method), 10

G

get_method() (tinyrpc.dispatch.RPCDispatcher method), 16
get_proxy() (tinyrpc.client.RPCClient method), 21

H

handle() (tinyrpc.transports.wsgi.WsgiServerTransport method), 19

I

InvalidReplyError, 22
InvalidRequestError, 22

J

JSONRPCProtocol (class in tinyrpc.protocols.jsonrpc), 13

M

MethodNotFoundError, 22

P

parse_reply() (tinyrpc.RPCProtocol method), 10
parse_request() (tinyrpc.RPCProtocol method), 10
public() (in module tinyrpc.dispatch), 17
public() (tinyrpc.dispatch.RPCDispatcher method), 16

R

receive_message() (tinyrpc.transports.callback.CallbackServerTransport method), 20
receive_message() (tinyrpc.transports.cgi.CGIServerTransport method), 19
receive_message() (tinyrpc.transports.ServerTransport method), 17
receive_one_message() (tinyrpc.server.RPCServer method), 21
register_instance() (tinyrpc.dispatch.RPCDispatcher method), 17
respond() (tinyrpc.RPCRequest method), 10
RPCBatchProtocol (class in tinyrpc), 11
RPCBatchRequest (class in tinyrpc), 11
RPCBatchResponse (class in tinyrpc), 11
RPCClient (class in tinyrpc.client), 20

RPCDispatcher (class in tinyrpc.dispatch), 15
RPCError, 22
RPCProtocol (class in tinyrpc), 9
RPCProxy (class in tinyrpc.client), 21
RPCRequest (class in tinyrpc), 10
RPCResponse (class in tinyrpc), 10
RPCServer (class in tinyrpc.server), 21

S

send_message() (tinyrpc.transports.ClientTransport method), 18
send_reply() (tinyrpc.transports.callback.CallbackServerTransport method), 20
send_reply() (tinyrpc.transports.cgi.CGIServerTransport method), 20
send_reply() (tinyrpc.transports.ServerTransport method), 17
serialize() (tinyrpc.RPCBatchResponse method), 11
serialize() (tinyrpc.RPCRequest method), 10
serialize() (tinyrpc.RPCResponse method), 10
serve_forever() (tinyrpc.server.RPCServer method), 22
ServerError, 22
ServerTransport (class in tinyrpc.transports), 17

T

tinyrpc.exc (module), 22
tinyrpc.server (module), 21
tinyrpc.server.gevent.RPCServerGreenlets (class in tinyrpc.server), 22

W

WsgiServerTransport (class in tinyrpc.transports.wsgi), 19

Z

ZmqClientTransport (class in tinyrpc.transports.zmq), 18
ZmqServerTransport (class in tinyrpc.transports.zmq), 18