tinyrpc Documentation

Release 1.1.0

Marc Brinkmann, Leo Noordergraaf

Aug 08, 2021

Contents

`tinyrpc` is a framework for constructing remote procedure call (RPC) services in Python.

In `tinyrpc` all components (transport, protocol and dispatcher) that together make an RPC service are independently replacable.

Although its initial scope is handling jsonrpc it is easy to add further protocols or add additional transports (one such example is msgpackrpc, which is now fully supported). If so desired it is even possible to replace the default method dispatcher.

CHAPTER 1

Table of contents

1.1 Quickstart examples

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

1.1.1 HTTP based

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

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

rpc_client = RPCClient(
    JSONRPCProtocol(),
    HttpPostClientTransport('http://localhost')
)

str_server = rpc_client.get_proxy()

# ...

# call a method called 'reverse_string' with a single string argument
result = str_server.reverse_string('Simple is better.')

print("Server answered:", result)
```

This call can be answered by a server implemented as follows:

```
import gevent
import gevent.pywsgi
import gevent.queue
```

(continues on next page)

(continued from previous page)

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

dispatcher = RPCDispatcher()
transport = WsgiServerTransport(queue_class=gevent.queue.Queue)

# start wsgi server as a background-greenlet
wsgi_server = gevent.pywsgi.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()
```

1.1.2 0mq

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

```
import zmq

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

ctx = zmq.Context()

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

str_server = rpc_client.get_proxy()

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

print("Server answered:", result)
```

Matching server:

```
import zmq

from tinyrpc.server import RPCServer
from tinyrpc.dispatch import RPCDispatcher
```

(continues on next page)

(continued from previous page)

```
from tinyrpc.protocols.jsonrpc import JSONRPCProtocol
from tinyrpc.transports.zmq import ZmqServerTransport

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()
```

1.1.3 Further examples

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

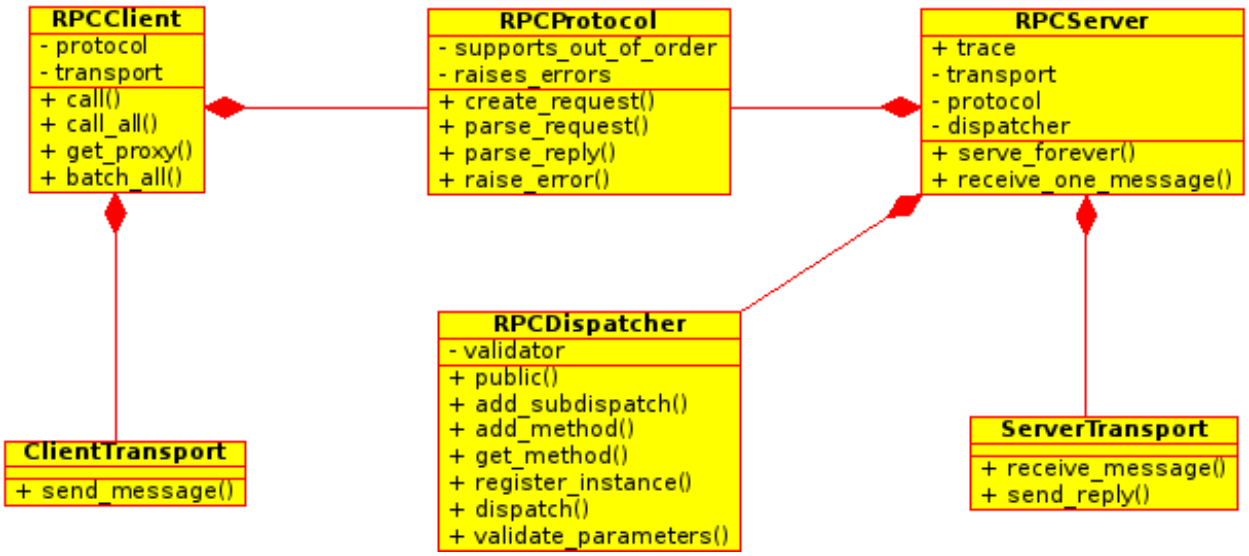
The *`RPCDispatcher`* should be useful on its own (or at least easily replaced with one of your choosing), see *Dispatching* for details.

1.2 Structure of tinyrpc

1.2.1 Architecture

`tinyrpc` is constructed around the `RPCServer` and `RPCClient` classes.

They in turn depend on the *`RPCDispatcher`*, *`RPCProtocol`*, `ServerTransport` and `ClientTransport` classes as visualized in the image below.



Of these *RPCProtocol*, ServerTransport and ClientTransport are abstract base classes.

Each layer is useful "on its own" and can be used separately. If you just need to decode a jsonrpc message, without passing it on or sending it through a transport, the *JSONRPCProtocol*-class is completely usable on its own.

Likewise the *RPCDispatcher* could be used to dispatch calls in a commandline REPL like application.

1.2.2 Transport

The transport classes are 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 as Python bytes objects.

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.

1.2.3 Protocol

The protocol class(es) are responsible for two tasks:

- they implement the protocol, defining how method names, method parameters and errors are represented in requests and responses.
- they serialize the requests and responses into messages and deserialize messages back into requests and responses.

1.2.4 Dispatcher

Dispatching performs the actual method calling determining with method to call and how to pass it the parameters. The result of the method call, or the exception if the call failed is assembled and made available to the protocol for serialization.

1.2.5 Client and Server

The client and server classes tie all components together to provide the application interface.

1.3 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.

1.3.1 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(comment_id)`.

These can be used with a dispatcher now:

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

(continues on next page)

(continued from previous page)

```
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
```

Class, static and unbound method dispatching

Although you will only rarely use these method types they *do* work and here we show you how.

Class methods do not have *self* as the initial parameter but rather a reference to their class. You may want to use such methods to instantiate class instances.

```
class ShowClassMethod:
    @classmethod
    @public
    def func(cls, a, b):
        return a-b
```

Note the ordering of the decorators. Ordering them differently will not work. You call dispatch to the *func* method just as you would dispatch to any other method.

Static methods have neither a class nor instance reference as first parameter:

```
class ShowStaticMethod:
    @staticmethod
    @public
    def func(a, b):
        return a-b
```

Again the ordering of the decorators is critical and you dispatch them as any other method.

Finally it is possible to dispatch to unbound methods but I strongly advise against it. If you really want to do that see the tests to learn how. Everyone else should use static methods instead.

1.3.2 API reference

class tinyrpc.dispatch.**RPCDispatcher**

Bases: object

Stores name-to-method mappings.

public(*name: str = None*) → Callable

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** (*str*) – Name to register callable with.

add_subdispatch(*dispatcher: tinyrpc.dispatch.RPCDispatcher*, *prefix: str = ''*)

Adds a subdispatcher, possibly in its own namespace.

Parameters

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

add_method(*f: Callable*, *name: str = None*) → None

Add a method to the dispatcher.

Parameters

- **f** (*callable*) – Callable to be added.

- **name** (*str*) – Name to register it with. If `None`, `f.__name__` will be used.

Raises ***RPCError*** – When the *name* is already registered.

get_method(*name: str*) → Callable
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.

Parameters **name** (*str*) – Function to find.

Returns The callable implementing the function.

Return type callable

Raises *MethodNotFoundError*

register_instance(*obj: object*, *prefix: str = ''*) → None
Create new subdispatcher and register all public object methods on it.

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

Parameters

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

dispatch(*request: Union[tinyrpc.protocols.RPCRequest, tinyrpc.protocols.RPCBatchRequest]*, *caller: Callable = None*) → Union[tinyrpc.protocols.RPCResponse, tinyrpc.protocols.RPCBatchResponse]
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 returned.

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.

Parameters

- **request** (`RPCRequest` *or* `RPCBatchRequest`) – The request containing the function to be called and its parameters.
- **caller** (*callable*) – An optional callable used to invoke the method.

Returns The result produced by calling the requested function.

Return type *RPCResponse* or *RPCBatchResponse*

Raises

- ***MethodNotFoundError*** – If the requested function is not published.
- ***ServerError*** – If some other error occurred.

Note: The *ServerError* is raised for any kind of exception not raised by the called function itself or *MethodNotFoundError*.

static validate_parameters(*method: Callable, args: List[Any], kwargs: Dict[str, Any]*) → None

Verify that **args* and ***kwargs* are appropriate parameters for *method*.

Warning: This function has changed to a static function. This will make it easier to replace it with a regular function instead of having to subclass only to replace it.

Parameters

- **method** – A callable.
- **args** – List of positional arguments for *method*
- **kwargs** – Keyword arguments for *method*

Raises ***InvalidParamsError*** – Raised when the provided arguments are not acceptable for *method*.

static validator(*method: Callable, args: List[Any], kwargs: Dict[str, Any]*) → None

Dispatched function parameter validation.

Type callable

By default this attribute is set to *validate_parameters()*. The value can be set to any callable implementing the same interface as *validate_parameters()* or to *None* to disable validation entirely.

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

tinyrpc.dispatch.**public**(*name: str = None*) → Callable

Decorator. Mark a method as eligible for registration by a dispatcher.

The dispatchers *register_instance()* function will do the actual registration of the marked method.

The difference with *public()* is that this decorator does not register with a dispatcher, therefore binding the marked methods with a dispatcher is delayed until runtime. It also becomes possible to bind with multiple dispatchers.

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

Example:

```
def class Baz(object):
    def not_exposed(self);
        # ...

    @public('do_something')
    def visible_method(self, arg1):
        # ...

baz = Baz()
dispatch = RPCDispatcher()
dispatch.register_instance(baz, 'bazzies`)
# Baz.visible_method is now callable via RPC as bazzies.do_something('hello')
```

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

1.4 The protocol layer

1.4.1 Interface definition

All protocols are implemented by deriving from *RPCProtocol* and implementing all of its members.

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.

Protocol-specific subclasses of *RPCRequest* and *RPCResponse* represent well-formed requests and responses.

Protocol specific subclasses of *RPCErrorResponse* represent errors and error 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.

API Reference

class tinyrpc.protocols.**RPCProtocol**

Bases: abc.ABC

Abstract base class for all protocol implementations.

supports_out_of_order = False

If true, this protocol can receive responses out of order correctly.

Note that this usually depends on the generation of unique_ids, the generation of these may or may not be thread safe, depending on the protocol. Ideally, only one instance of RPCProtocol should be used per client.

Type bool

raises_errors = True

If True, this protocol instance will raise an RPCError exception.

On receipt of an RPCErrorResponse instance an RPCError exception is raised. When this flag is False the RPCErrorResponse object is returned to the caller which is then responsible for handling the error.

Type bool

create_request(*method: str*, *args: List[Any] = None*, *kwargs: Dict[str, Any] = None*, *one_way: bool = False*) → tinyrpc.protocols.RPCRequest

Creates a new *RPCRequest* object.

Called by the client when constructing a request. 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** (*str*) – The method name to invoke.
- **args** (*list*) – The positional arguments to call the method with.
- **kwargs** (*dict*) – The keyword arguments to call the method with.
- **one_way** (*bool*) – The request is an update, i.e. it does not expect a reply.

Returns A new request instance

Return type *RPCRequest*

parse_request(*data: bytes*) → tinyrpc.protocols.RPCRequest
De-serializes and validates a request.

Called by the server to reconstruct the serialized *RPCRequest*.

Parameters **data** (*bytes*) – The data stream received by the transport layer containing the serialized request.

Returns A reconstructed request.

Return type *RPCRequest*

parse_reply(*data: bytes*) → tinyrpc.protocols.RPCResponse
De-serializes and validates a response.

Called by the client to reconstruct the serialized *RPCResponse*.

Parameters **data** (*bytes*) – The data stream received by the transport layer containing the serialized response.

Returns A reconstructed response.

Return type *RPCResponse*

raise_error(*error: tinyrpc.protocols.RPCErrorResponse*) → tinyrpc.exc.RPCError
Raises the exception in the client.

Called by the client to convert the *RPCErrorResponse* into an Exception and raise or return it depending on the *raises_errors* attribute.

Parameters **error** (*RPCResponse*) – The error response received from the server.

Return type *RPCError* when *raises_errors* is False.

Raises *RPCError* when *raises_errors* is True.

class tinyrpc.protocols.**RPCRequest**
Bases: object

Defines a generic RPC request.

unique_id = None
Correlation ID used to match request and response.

Type int or str or None

Protocol specific, may or may not be set. This value should only be set by *create_request()*.

When the protocol permits it this ID allows servers to respond to requests out of order and allows clients to relate a response to the corresponding request.

Only supported if the protocol has its *supports_out_of_order* set to True.

Generated by the client, the server copies it from request to corresponding response.

method = None
The name of the RPC function to be called.

Type str

The *method* attribute uses the name of the function as it is known by the public. The *RPCDispatcher* allows the use of public aliases in the @public decorators. These are the names used in the *method* attribute.

args = None
The positional arguments of the method call.

Type list

The contents of this list are the positional parameters for the *method* called. It is eventually called as `method(*args)`.

kwargs = None
The keyword arguments of the method call.

Type dict

The contents of this dict are the keyword parameters for the *method* called. It is eventually called as `method(**kwargs)`.

error_respond(*error: Union[Exception, str]*) → Optional[tinyrpc.protocols.RPCErrorResponse]
Creates an error response.

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

This is an abstract method that must be overridden in a derived class.

Parameters **error** (*Exception or str*) – An exception or a string describing the error.

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

Return type *RPCErrorResponse*

respond(*result: Any*) → Optional[tinyrpc.protocols.RPCResponse]
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`.

This is an abstract method that must be overridden in a derived class.

Parameters **result** (*Any type that can be serialized by the protocol.*) – Passed on to new response instance.

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

Return type *RPCResponse*

serialize() → bytes
Returns a serialization of the request.

Converts the request into a bytes object that can be passed to and by the transport layer.

This is an abstract method that must be overridden in a derived class.

Returns A bytes object to be passed on to a transport.

Return type bytes

class tinyrpc.protocols.**RPCResponse**
Bases: abc.ABC

Defines a generic RPC response.

Base class for all responses.

id
Correlation ID to match request and response

Type str or int

result
When present this attribute contains the result of the RPC call. Otherwise the *error* attribute must be defined.

Type Any type that can be serialized by the protocol.

error
When present the *result* attribute must be absent. Presence of this attribute indicates an error condition.

Type *RPCError*

unique_id = None
Correlation ID used to match request and response.

Type int or str or None

serialize() → bytes
Returns a serialization of the response.

Converts the response into a bytes object that can be passed to and by the transport layer.

This is an abstract method that must be overridden in a derived class.

Returns The serialized encoded response object.

Return type bytes

class tinyrpc.protocols.**RPCErrorResponse**
Bases: *tinyrpc.protocols.RPCResponse*, abc.ABC

RPC error response class.

Base class for all deriving responses.

error
This attribute contains the fields `message` (str) and `code` (int) where at least `message` is required to contain a value.

Type dict

class tinyrpc.exc.**BadRequestError**
Bases: *tinyrpc.exc.RPCError*, abc.ABC

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

1.4.2 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.

API Reference

class tinyrpc.protocols.**RPCBatchProtocol**
Bases: *tinyrpc.protocols.RPCProtocol*, abc.ABC

Abstract base class for all batch protocol implementations.

create_batch_request(*requests: List[RPCRequest] = None*) → tinyrpc.protocols.RPCBatchRequest
Create a new *RPCBatchRequest* object.

Called by the client when constructing a request.

Parameters **requests** (list or *RPCRequest*) – A list of requests.

Returns A new request instance.

Return type *RPCBatchRequest*

class tinyrpc.protocols.**RPCBatchRequest**
Bases: list

Multiple requests batched together.

Protocols that support multiple requests in a single message use this to group them together. Note that not all protocols may support batch requests.

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 an *RPCRequest* or an *BadRequestError*, which indicates that there has been an error in parsing the request.

create_batch_response() → Optional[tinyrpc.protocols.RPCBatchResponse]
Creates a response suitable for responding to this request.

This is an abstract method that must be overridden in a derived class.

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

Return type *RPCBatchResponse*

serialize() → bytes
Returns a serialization of the request.

Converts the request into a bytes object that can be passed to and by the transport layer.

This is an abstract method that must be overridden in a derived class.

Returns A bytes object to be passed on to a transport.

Return type bytes

class tinyrpc.protocols.**RPCBatchResponse**
Bases: list

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() → bytes
Returns a serialization of the batch response.

Converts the response into a bytes object that can be passed to and by the transport layer.

This is an abstract method that must be overridden in a derived class.

Returns A bytes object to be passed on to a transport.

Return type bytes

1.4.3 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`.

1.5 The JSON-RPC protocol

1.5.1 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...
```

(continues on next page)

(continued from previous page)

```
        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
```

1.5.2 Protocol implementation

API Reference

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

JSONRPC protocol implementation.

JSON_RPC_VERSION = '2.0'
Currently, only version 2.0 is supported.

request_factory() → tinyrpc.protocols.jsonrpc.JSONRPCRequest
Factory for request objects.

Allows derived classes to use requests derived from *JSONRPCRequest*.

Return type *JSONRPCRequest*

create_batch_request(*requests: Union[JSONRPCRequest, List[JSONRPCRequest]] = None*) → tinyrpc.protocols.jsonrpc.JSONRPCBatchRequest
Create a new *JSONRPCBatchRequest* object.

Called by the client when constructing a request.

Parameters **requests** (list or *JSONRPCRequest*) – A list of requests.

Returns A new request instance.

Return type *JSONRPCBatchRequest*

create_request(*method: str*, *args: List[Any] = None*, *kwargs: Dict[str, Any] = None*, *one_way: bool = False*) → tinyrpc.protocols.jsonrpc.JSONRPCRequest
Creates a new *JSONRPCRequest* object.

Called by the client when constructing a request. JSON RPC allows either the args or kwargs argument to be set.

Parameters

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

Returns A new request instance

Return type *JSONRPCRequest*

Raises ***InvalidRequestError*** – when args and kwargs are both defined.

parse_reply(*data: bytes*) → Union[tinyrpc.protocols.jsonrpc.JSONRPCSuccessResponse, tinyrpc.protocols.jsonrpc.JSONRPCErrorResponse]
De-serializes and validates a response.

Called by the client to reconstruct the serialized JSONRPCResponse.

Parameters **data** (*bytes*) – The data stream received by the transport layer containing the serialized request.

Returns A reconstructed response.

Return type *JSONRPCSuccessResponse* or *JSONRPCErrorResponse*

Raises ***InvalidReplyError*** – if the response is not valid JSON or does not conform to the standard.

parse_request(*data: bytes*) → Union[tinyrpc.protocols.jsonrpc.JSONRPCRequest, tinyrpc.protocols.jsonrpc.JSONRPCBatchRequest]

De-serializes and validates a request.

Called by the server to reconstruct the serialized *JSONRPCRequest*.

Parameters **data** (*bytes*) – The data stream received by the transport layer containing the serialized request.

Returns A reconstructed request.

Return type *JSONRPCRequest*

Raises

- ***JSONRPCParseError*** – if the `data` cannot be parsed as valid JSON.
- ***JSONRPCInvalidRequestError*** – if the request does not comply with the standard.

raise_error(*error: Union[JSONRPCErrorResponse, Dict[str, Any]]*) → tinyrpc.protocols.jsonrpc.JSONRPCError

Recreates the exception.

Creates a *JSONRPCError* instance and raises it. This allows the error, message and data attributes of the original exception to propagate into the client code.

The `raises_error` flag controls if the exception object is raised or returned.

Returns the exception object if it is not allowed to raise it.

Raises ***JSONRPCError*** – when the exception can be raised. The exception object will contain `message`, `code` and optionally a `data` property.

class tinyrpc.protocols.jsonrpc.**JSONRPCRequest**

Bases: *tinyrpc.protocols.RPCRequest*

Defines a JSON RPC request.

one_way = None

Request or Notification.

Type bool

This flag indicates if the client expects to receive a reply (request: `one_way = False`) or not (notification: `one_way = True`).

Note that according to the specification it is possible for the server to return an error response. For example if the request becomes unreadable and the server is not able to determine that it is in fact a notification an error should be returned. However, once the server had verified that the request is a notification no reply (not even an error) should be returned.

unique_id = None

Correlation ID used to match request and response.

Type int or str

Generated by the client, the server copies it from request to corresponding response.

method = None

The name of the RPC function to be called.

Type str

The *method* attribute uses the name of the function as it is known by the public. The *RPCDispatcher* allows the use of public aliases in the `@public` decorators. These are the names used in the *method* attribute.

args = None
The positional arguments of the method call.

Type list

The contents of this list are the positional parameters for the *method* called. It is eventually called as `method(*args)`.

kwargs = None
The keyword arguments of the method call.

Type dict

The contents of this dict are the keyword parameters for the *method* called. It is eventually called as `method(**kwargs)`.

error_respond(*error: Union[Exception, str]*) → Optional[tinyrpc.protocols.jsonrpc.JSONRPCErrorResponse]
Create an error response to this request.

When processing the request produces an error condition this method can be used to create the error response object.

Parameters **error** (*Exception or str*) – Specifies what error occurred.

Returns An error response object that can be serialized and sent to the client.

Return type ;py:class:*JSONRPCErrorResponse*

respond(*result: Any*) → Optional[tinyrpc.protocols.jsonrpc.JSONRPCSuccessResponse]
Create a response to this request.

When processing the request completed successfully this method can be used to create a response object.

Parameters **result** (*Anything that can be encoded by JSON.*) – The result of the invoked method.

Returns A response object that can be serialized and sent to the client.

Return type *JSONRPCSuccessResponse*

serialize() → bytes
Returns a serialization of the request.

Converts the request into a bytes object that can be sent to the server.

Returns The serialized encoded request object.

Return type bytes

class tinyrpc.protocols.jsonrpc.**JSONRPCSuccessResponse**
Bases: *tinyrpc.protocols.RPCResponse*

Collects the attributes of a successful response message.

Contains the fields of a normal (i.e. a non-error) response message.

unique_id
Correlation ID to match request and response. A JSON RPC response *must* have a defined matching id attribute. `None` is not a valid value for a successful response.

Type str or int

result
Contains the result of the RPC call.

Type Any type that can be serialized by the protocol.

serialize() → bytes
Returns a serialization of the response.

Converts the response into a bytes object that can be passed to and by the transport layer.

Returns The serialized encoded response object.

Return type bytes

class tinyrpc.protocols.jsonrpc.**JSONRPCErrorResponse**
Bases: *tinyrpc.protocols.RPCErrorResponse*

Collects the attributes of an error response message.

Contains the fields of an error response message.

unique_id
Correlation ID to match request and response. None is a valid ID when the error cannot be matched to a particular request.

Type str or int or None

error
The error message. A string describing the error condition.

Type str

data
This field may contain any JSON encodable datum that the server may want to return the client.

It may contain additional information about the error condition, a partial result or whatever. Its presence and value are entirely optional.

Type Any type that can be serialized by the protocol.

_jsonrpc_error_code
The numeric error code.

The value is usually predefined by one of the JSON protocol exceptions. It can be set by the developer when defining application specific exceptions. See *FixedErrorMessageMixin* for an example on how to do this.

Note that the value of this field *must* comply with the defined values in the standard.

serialize() → bytes
Returns a serialization of the error.

Converts the response into a bytes object that can be passed to and by the transport layer.

Returns The serialized encoded error object.

Return type bytes

1.5.3 Batch protocol

API Reference

class tinyrpc.protocols.jsonrpc.**JSONRPCBatchRequest**
Bases: *tinyrpc.protocols.RPCBatchRequest*

Defines a JSON RPC batch request.

create_batch_response() → Optional[tinyrpc.protocols.jsonrpc.JSONRPCBatchResponse]
Produces a batch response object if a response is expected.

Returns A batch response if needed

Return type *JSONRPCBatchResponse*

serialize() → bytes
Returns a serialization of the request.

Converts the request into a bytes object that can be passed to and by the transport layer.

Returns A bytes object to be passed on to a transport.

Return type bytes

class tinyrpc.protocols.jsonrpc.**JSONRPCBatchResponse**
Bases: *tinyrpc.protocols.RPCBatchResponse*

Multiple responses from a batch request. See *JSONRPCBatchRequest* on how to handle.

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

serialize() → bytes
Returns a serialization of the batch response.

Converts the response into a bytes object that can be passed to and by the transport layer.

Returns A bytes object to be passed on to a transport.

Return type bytes

1.5.4 Errors and error handling

API Reference

class tinyrpc.protocols.jsonrpc.**FixedErrorMessageMixin**(**args*, ***kwargs*)
Bases: object

Combines JSON RPC exceptions with the generic RPC exceptions.

Constructs the exception using the provided parameters as well as properties of the JSON RPC Exception.

JSON RPC exceptions declare two attributes:

jsonrpc_error_code
This is an error code conforming to the JSON RPC error codes convention.

Type int

message
This is a textual representation of the error code.

Type str

Parameters

- **args** (*list*) – Positional arguments for the constructor. When present it overrules the *message* attribute.
- **kwargs** (*dict*) – Keyword arguments for the constructor. If the data parameter is found in kwargs its contents are used as the *data* property of the JSON RPC Error object.

FixedErrorMessageMixin is the basis for adding your own exceptions to the predefined ones. Here is a version of the reverse string example that dislikes palindromes:

```
class PalindromeError(FixedErrorMessageMixin, Exception)
    jsonrpc_error_code = 99
    message = "Ah, that's cheating"

@public
def reverse_string(s):
    r = s[::-1]
    if r == s:
        raise PalindromeError(data=s)
    return r
```

```
>>> client.reverse('rotator')
```

Will return an error object to the client looking like:

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "error": {
        "code": 99,
        "message": "Ah, that's cheating",
        "data": "rotator"
    }
}
```

error_respond() → tinyrpc.protocols.jsonrpc.JSONRPCErrorResponse
Converts the error to an error response object.

Returns An error response object ready to be serialized and sent to the client.

Return type *JSONRPCErrorResponse*

class tinyrpc.protocols.jsonrpc.**JSONRPCParseError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.jsonrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

The request cannot be decoded or is malformed.

class tinyrpc.protocols.jsonrpc.**JSONRPCInvalidRequestError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.jsonrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

The request contents are not valid for JSON RPC 2.0

class tinyrpc.protocols.jsonrpc.**JSONRPCMethodNotFoundError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.jsonrpc.FixedErrorMessageMixin*, *tinyrpc.exc.MethodNotFoundError*

The requested method name is not found in the registry.

class tinyrpc.protocols.jsonrpc.**JSONRPCInvalidParamsError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.jsonrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

The provided parameters are not appropriate for the function called.

class tinyrpc.protocols.jsonrpc.**JSONRPCInternalError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.jsonrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

Unspecified error, not in the called function.

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

Bases: *tinyrpc.protocols.jsonrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

Unspecified error, this message originates from the called function.

class tinyrpc.protocols.jsonrpc.**JSONRPCError**(*error: Union[JSONRPCErrorResponse, Dict[str, Any]]*)

Bases: *tinyrpc.protocols.jsonrpc.FixedErrorMessageMixin*, *tinyrpc.exc.RPCError*

Reconstructs (to some extend) the server-side exception.

The client creates this exception by providing it with the `error` attribute of the JSON error response object returned by the server.

Parameters **error** (*dict*) – This dict contains the error specification:

- code (int): the numeric error code.
- message (str): the error description.
- data (any): if present, the data attribute of the error

1.5.5 Adding custom exceptions

Note: As per the specification you should use error codes -32000 to -32099 when adding server specific error messages. Error codes outside the range -32768 to -32000 are available for application specific error codes.

To add custom errors you need to combine an `Exception` subclass with the *FixedErrorMessageMixin* class to create your exception object which you can raise.

So a version of the reverse string example that dislikes palindromes could look like:

```
from tinyrpc.protocols.jsonrpc import FixedErrorMessageMixin, JSONRPCProtocol
from tinyrpc.dispatch import RPCDispatcher

dispatcher = RPCDispatcher()

class PalindromeError(FixedErrorMessageMixin, Exception):
    jsonrpc_error_code = 99
    message = "Ah, that's cheating!"


@dispatcher.public
def reverse_string(s):
    r = s[::-1]
    if r == s:
        raise PalindromeError()
    return r
```

1.5.6 Error with data

The specification states that the `error` element of a reply may contain an optional `data` property. This property is now available for your use.

There are two ways that you can use to pass additional data with an `Exception`. It depends whether your application generates regular exceptions or exceptions derived from *FixedErrorMessageMixin*.

When using ordinary exceptions you normally pass a single parameter (an error message) to the `Exception` constructor. By passing two parameters, the second parameter is assumed to be the data element.

```
@public
def fn():
    raise Exception('error message', {'msg': 'structured data', 'lst': [1, 2, 3]})
```

This will produce the reply message:

```
{   "jsonrpc": "2.0",
    "id": <some id>,
    "error": {
        "code": -32000,
        "message": "error message",
        "data": {"msg": "structured data", "lst": [1, 2, 3]}
    }
}
```

When using *`FixedErrorMessageMixin`* based exceptions the data is passed using a keyword parameter.

```
class MyException(FixedErrorMessageMixin, Exception):
    jsonrcp_error_code = 99
    message = 'standard message'

@public
def fn():
    raise MyException(data={'msg': 'structured data', 'lst': [1, 2, 3]})
```

This will produce the reply message:

```
{   "jsonrpc": "2.0",
    "id": <some id>,
    "error": {
        "code": 99,
        "message": "standard message",
        "data": {"msg": "structured data", "lst": [1, 2, 3]}
    }
}
```

1.6 The MSGPACK-RPC protocol

1.6.1 Example

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

Server

```
from tinyrpc.protocols.msgpackrpc import MSGPACKRPCProtocol
from tinyrpc import BadRequestError, RPCRequest

rpc = MSGPACKRPCProtocol()
```

(continues on next page)

(continued from previous page)

```
# the code below is valid for all protocols, not just MSGPACKRPCProtocol,
# as long as you don't need to handle batch RPC requests:

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
        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.msgpackrpc import MSGPACKRPCProtocol

rpc = MSGPACKRPCProtocol()

# 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)
```

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
```

1.6.2 Protocol implementation

API Reference

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCProtocol**(**args*, ***kwargs*)

Bases: *tinyrpc.protocols.RPCProtocol*

MSGPACKRPC protocol implementation.

request_factory() → tinyrpc.protocols.msgpackrpc.MSGPACKRPCRequest

Factory for request objects.

Allows derived classes to use requests derived from *MSGPACKRPCRequest*.

Return type *MSGPACKRPCRequest*

create_request(*method: str*, *args: List[Any] = None*, *kwargs: Dict[str, Any] = None*, *one_way: bool = False*) → tinyrpc.protocols.msgpackrpc.MSGPACKRPCRequest

Creates a new *MSGPACKRPCRequest* object.

Called by the client when constructing a request. MSGPACK-RPC allows only the `args` argument to be set; keyword arguments are not supported.

Parameters

- **method** (*str*) – The method name to invoke.
- **args** (*list*) – The positional arguments to call the method with.
- **kwargs** (*dict*) – The keyword arguments to call the method with; must be `None` as the protocol does not support keyword arguments.
- **one_way** (*bool*) – The request is an update, i.e. it does not expect a reply.

Returns A new request instance

Return type *MSGPACKRPCRequest*

Raises ***InvalidRequestError*** – when `kwargs` is defined.

parse_reply(*data: bytes*) → Union[tinyrpc.protocols.msgpackrpc.MSGPACKRPCSuccessResponse, tinyrpc.protocols.msgpackrpc.MSGPACKRPCErrorResponse]

De-serializes and validates a response.

Called by the client to reconstruct the serialized `MSGPACKRPCResponse`.

Parameters **data** (*bytes*) – The data stream received by the transport layer containing the serialized response.

Returns A reconstructed response.

Return type *MSGPACKRPCSuccessResponse* or *MSGPACKRPCErrorResponse*

Raises ***InvalidReplyError*** – if the response is not valid MSGPACK or does not conform to the standard.

parse_request(*data: bytes*) → tinyrpc.protocols.msgpackrpc.MSGPACKRPCRequest

De-serializes and validates a request.

Called by the server to reconstruct the serialized *MSGPACKRPCRequest*.

Parameters **data** (*bytes*) – The data stream received by the transport layer containing the serialized request.

Returns A reconstructed request.

Return type *MSGPACKRPCRequest*

Raises

- ***MSGPACKRPCParseError*** – if the `data` cannot be parsed as valid MSGPACK.
- ***MSGPACKRPCInvalidRequestError*** – if the request does not comply with the standard.

raise_error(*error: Union[MSGPACKRPCErrorResponse, Dict[str, Any]]*) → tinyrpc.protocols.msgpackrpc.MSGPACKRPCError

Recreates the exception.

Creates a *MSGPACKRPCError* instance and raises it.

This allows the error code and the message of the original exception to propagate into the client code.

The `raises_error` flag controls if the exception object is raised or returned.

Returns the exception object if it is not allowed to raise it.

Raises ***MSGPACKRPCError*** – when the exception can be raised. The exception object will contain `message` and `code`.

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCRequest**

Bases: *tinyrpc.protocols.RPCRequest*

Defines a MSGPACK-RPC request.

one_way = None

Request or Notification.

Type bool

This flag indicates if the client expects to receive a reply (request: `one_way = False`) or not (notification: `one_way = True`).

Note that it is possible for the server to return an error response. For example if the request becomes unreadable and the server is not able to determine that it is in fact a notification an error should be returned. However, once the server had verified that the request is a notification no reply (not even an error) should be returned.

unique_id = None

Correlation ID used to match request and response.

Type int

Generated by the client, the server copies it from request to corresponding response.

method = None

The name of the RPC function to be called.

Type str

The *method* attribute uses the name of the function as it is known by the public. The *RPCDispatcher* allows the use of public aliases in the `@public` decorators. These are the names used in the *method* attribute.

args = None

The positional arguments of the method call.

Type list

The contents of this list are the positional parameters for the *method* called. It is eventually called as `method(*args)`.

error_respond(*error: Union[Exception, str]*) → Optional[tinyrpc.protocols.msgpackrpc.MSGPACKRPCErrorResponse]
Create an error response to this request.

When processing the request produces an error condition this method can be used to create the error response object.

Parameters **error** (*Exception or str*) – Specifies what error occurred.

Returns An error response object that can be serialized and sent to the client.

Return type ;py:class:*MSGPACKRPCErrorResponse*

respond(*result: Any*) → Optional[tinyrpc.protocols.msgpackrpc.MSGPACKRPCSuccessResponse]
Create a response to this request.

When processing the request completed successfully this method can be used to create a response object.

Parameters **result** (*Anything that can be encoded by MSGPACK.*) – The result of the invoked method.

Returns A response object that can be serialized and sent to the client.

Return type *MSGPACKRPCSuccessResponse*

serialize() → bytes
Returns a serialization of the request.

Converts the request into a bytes object that can be passed to and by the transport layer.

This is an abstract method that must be overridden in a derived class.

Returns A bytes object to be passed on to a transport.

Return type bytes

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCSuccessResponse**
Bases: *tinyrpc.protocols.RPCResponse*

serialize()
Returns a serialization of the response.

Converts the response into a bytes object that can be passed to and by the transport layer.

This is an abstract method that must be overridden in a derived class.

Returns The serialized encoded response object.

Return type bytes

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCErrorResponse**
Bases: *tinyrpc.protocols.RPCErrorResponse*

serialize()
Returns a serialization of the response.

Converts the response into a bytes object that can be passed to and by the transport layer.

This is an abstract method that must be overridden in a derived class.

Returns The serialized encoded response object.

Return type bytes

1.6.3 Errors and error handling

API Reference

class tinyrpc.protocols.msgpackrpc.**FixedErrorMessageMixin**(**args*, ***kwargs*)
Bases: object

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCParseError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.msgpackrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCInvalidRequestError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.msgpackrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCMethodNotFoundError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.msgpackrpc.FixedErrorMessageMixin*, *tinyrpc.exc.MethodNotFoundError*

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCInvalidParamsError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.msgpackrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCInternalError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.msgpackrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCServerError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.msgpackrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCError**(*error: Union[MSGPACKRPCErrorResponse, Tuple[int, str]]*)
Bases: *tinyrpc.protocols.msgpackrpc.FixedErrorMessageMixin*, *tinyrpc.exc.RPCError*

Reconstructs (to some extend) the server-side exception.

The client creates this exception by providing it with the `error` attribute of the MSGPACK error response object returned by the server.

Parameters **error** – This tuple contains the error specification: the numeric error code and the error description.

1.6.4 Adding custom exceptions

Note: Unlike JSON-RPC, the MSGPACK-RPC specification does not specify how the error messages should look like; the protocol allows any arbitrary MSGPACK object as an error object. For sake of compatibility with JSON-RPC, this implementation uses MSGPACK lists of length 2 (consisting of a numeric error code and an error description) to represent errors in the serialized representation. These are transparently decoded into *MSGPACKRPCError* instances as needed. The error codes for parsing errors, invalid requests, unknown RPC methods and so on match those from the JSON-RPC specification.

To add custom errors you need to combine an `Exception` subclass with the *`FixedErrorMessageMixin`* class to create your exception object which you can raise.

So a version of the reverse string example that dislikes palindromes could look like:

```
from tinyrpc.protocols.msgpackrpc import FixedErrorMessageMixin, MSGPACKRPCProtocol
from tinyrpc.dispatch import RPCDispatcher

dispatcher = RPCDispatcher()

class PalindromeError(FixedErrorMessageMixin, Exception):
    msgpackrpc_error_code = 99
    message = "Ah, that's cheating!"


@dispatcher.public
def reverse_string(s):
    r = s[::-1]
    if r == s:
        raise PalindromeError()
    return r
```

1.7 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**

Bases: `object`

Abstract base class for all server transports.

The server side implementation of the transport component. Requests and replies encoded by the protocol component are exchanged between client and server using the `ServerTransport` and `ClientTransport` classes.

receive_message() → Tuple[Any, bytes]

Receive a message from the transport.

Blocks until a message has been received. May return an opaque context object to its caller that should be passed on to `send_reply()` to identify the transport or requester later on. Use and function of the context object are entirely controlled by the transport instance.

The message must be treated as a binary entity as only the protocol level will know how to interpret the message.

If the transport encodes the message in some way, the opposite end is responsible for decoding it before it is passed to either client or server.

Returns A tuple consisting of `(context, message)`. Where `context` can be any valid Python type and `message` must be a `bytes` object.

send_reply(*context: Any*, *reply: bytes*) → None

Sends a reply to a client.

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

The reply must be a bytes object since only the protocol level will know how to construct the reply.

Parameters

- **context** (*any*) – A context returned by `receive_message()`.
- **reply** (*bytes*) – The reply to return to the client.

class tinyrpc.transports.**ClientTransport**
Bases: object

Abstract base class for all client transports.

The client side implementation of the transport component. Requests and replies encoded by the protocol component are exchanged between client and server using the `ServerTransport` and `ClientTransport` classes.

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

Sends a message to the connected server.

The message must be treated as a binary entity as only the protocol level will know how to interpret the message.

If the transport encodes the message in some way, the opposite end is responsible for decoding it before it is passed to either client or server.

This function will block until the reply has been received.

Parameters

- **message** (*bytes*) – The request to send to the server.
- **expect_reply** (*bool*) – Some protocols allow notifications for which a reply is not expected. When this flag is `False` the transport may not wait for a response from the server. **Note** that it is still the responsibility of the transport layer how to implement this. It is still possible that the server sends some form of reply regardless the value of this flag.

Returns The servers reply to the request.

Return type bytes

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.

Also note that the client transport interface is not designed for asynchronous use. For simple use cases (sending multiple concurrent requests) monkey patching with gevent may get the job done.

1.7.1 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: zmq.sugar.socket.Socket*)
Bases: tinyrpc.transports.ServerTransport

Server transport based on a `zmq.ROUTER` socket.

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

receive_message() → Tuple[Any, bytes]
Receive a message from the transport.

Blocks until a message has been received. May return an opaque context object to its caller that should be passed on to `send_reply()` to identify the transport or requester later on. Use and function of the context object are entirely controlled by the transport instance.

The message must be treated as a binary entity as only the protocol level will know how to interpret the message.

If the transport encodes the message in some way, the opposite end is responsible for decoding it before it is passed to either client or server.

Returns A tuple consisting of `(context, message)`. Where `context` can be any valid Python type and `message` must be a `bytes` object.

send_reply(*context: Any*, *reply: bytes*) → None
Sends a reply to a client.

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

The reply must be a bytes object since only the protocol level will know how to construct the reply.

Parameters

- **context** (*any*) – A context returned by `receive_message()`.
- **reply** (*bytes*) – The reply to return to the client.

classmethod create(*zmq_context: zmq.sugar.context.Context*, *endpoint: str*) → tinyrpc.transports.zmq.ZmqServerTransport
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: zmq.sugar.socket.Socket*)
Bases: `tinyrpc.transports.ClientTransport`

Client transport based on a `zmq.REQ` socket.

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

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

Sends a message to the connected server.

The message must be treated as a binary entity as only the protocol level will know how to interpret the message.

If the transport encodes the message in some way, the opposite end is responsible for decoding it before it is passed to either client or server.

This function will block until the reply has been received.

Parameters

- **message** (*bytes*) – The request to send to the server.
- **expect_reply** (*bool*) – Some protocols allow notifications for which a reply is not expected. When this flag is `False` the transport may not wait for a response from the server. **Note** that it is still the responsibility of the transport layer how to implement this. It is still possible that the server sends some form of reply regardless the value of this flag.

Returns The servers reply to the request.

Return type bytes

classmethod create(*zmq_context: zmq.sugar.context.Context*, *endpoint: str*) → tinyrpc.transports.zmq.ZmqClientTransport

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).

class tinyrpc.transports.http.**HttpPostClientTransport**(*endpoint: str*, *post_method: Callable = None*, ***kwargs*)

Bases: `tinyrpc.transports.ClientTransport`

HTTP POST based client transport.

Requires `requests`. Submits messages to a server using the body of an `HTTP POST` request. Replies are taken from the responses body.

Parameters

- **endpoint** (*str*) – The URL to send `POST` data to.
- **post_method** (*callable*) – allows to replace *requests.post* with another method, e.g. the post method of a *requests.Session()* instance.
- **kwargs** (*dict*) – Additional parameters for `requests.post()`.

send_message(*message: bytes*, *expect_reply: bool = True*)

Send a message to the server and possibly receive a reply.

Sends a message to the connected server.

The message must be treated as a binary entity as only the protocol level will know how to interpret the message.

If the transport encodes the message in some way, the opposite end is responsible for decoding it before it is passed to either client or server.

This function will block until the reply has been received.

Parameters

- **message** (*bytes*) – The request to send to the server.

- **expect_reply** (*bool*) – Some protocols allow notifications for which a reply is not expected. When this flag is `False` the transport may not wait for a response from the server. **Note** that it is still the responsibility of the transport layer how to implement this. It is still possible that the server sends some form of reply regardless the value of this flag.

Returns The servers reply to the request.

Return type bytes

Note: To set a timeout on your client transport provide a `timeout` keyword parameter like:

```
transport = HttpPostClientTransport(endpoint, timeout=0.1)
```

It will result in a `requests.exceptions.Timeout` exception when a timeout occurs.

WSGI

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

Bases: `tinyrpc.transports.ServerTransport`

WSGI transport.

Requires `werkzeug`.

Due to the nature of WSGI, this transport has a few peculiarities: 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).

receive_message() → Tuple[Any, bytes]

Receive a message from the transport.

Blocks until a message has been received. May return an opaque context object to its caller that should be passed on to `send_reply()` to identify the transport or requester later on. Use and function of the context object are entirely controlled by the transport instance.

The message must be treated as a binary entity as only the protocol level will know how to interpret the message.

If the transport encodes the message in some way, the opposite end is responsible for decoding it before it is passed to either client or server.

Returns A tuple consisting of `(context, message)`. Where `context` can be any valid Python type and `message` must be a `bytes` object.

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

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

The reply must be a bytes object since only the protocol level will know how to construct the reply.

Parameters

- **context** (*any*) – A context returned by `receive_message()`.
- **reply** (*bytes*) – The reply to return to the client.

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**
Bases: `tinyrpc.transports.ServerTransport`

CGI transport.

The CGIServerTransport adds CGI as a supported server protocol. It can be used with the regular HTTP client.

Reading stdin is blocking but, given that we've been called, something is waiting. The transport accepts only POST requests.

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

receive_message() → Tuple[Any, bytes]
Receive a message from the transport.

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

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

send_reply(*context: Any*, *reply: bytes*) → None
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 bytes, it is up to the sender to convert the message beforehand. A non-bytes value raises a `TypeError`.

Parameters

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

Callback

class tinyrpc.transports.callback.**CallbackServerTransport**(*reader: Callable[[], bytes], writer: Callable[[bytes], None]*)

Bases: tinyrpc.transports.ServerTransport

Callback server transport.

The CallbackServerTransport uses the provided callbacks to implement communication with the counterpart.

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 are set when constructed.

Parameters

- **reader** (*callable*) – Called when the transport wants to receive a new request.

  returns The RPC request.

  rtype bytes

- **writer(reply)** (*callable*) – Called to return the response to the client.

  param bytes reply The response to the request.

receive_message() → Tuple[Any, bytes]
Receive a message from the transport.

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

Returns A tuple consisting of (context, message).

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

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

Uses the callback function writer to forward the reply.

Parameters

- **context** (*any*) – A context returned by receive_message().
- **reply** (*bytes*) – The reply.

RabbitMQ

class tinyrpc.transports.rabbitmq.**RabbitMQServerTransport**(*connection: pika.adapters.blocking_connection.BlockingCon queue: str, exchange: str = ''*)

Bases: tinyrpc.transports.ServerTransport

Server transport based on a pika.BlockingConnection.

The transport assumes a RabbitMQ topology has already been established.

Parameters

- **connection** – A `pika.BlockingConnection` instance.
- **queue** – The RabbitMQ queue to consume messages from.
- **exchange** – The RabbitMQ exchange to use.

receive_message() → Tuple[Any, bytes]
Receive a message from the transport.

Blocks until a message has been received. May return an opaque context object to its caller that should be passed on to `send_reply()` to identify the transport or requester later on. Use and function of the context object are entirely controlled by the transport instance.

The message must be treated as a binary entity as only the protocol level will know how to interpret the message.

If the transport encodes the message in some way, the opposite end is responsible for decoding it before it is passed to either client or server.

Returns A tuple consisting of `(context, message)`. Where `context` can be any valid Python type and `message` must be a `bytes` object.

send_reply(*context: Any*, *reply: bytes*) → None
Sends a reply to a client.

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

The reply must be a bytes object since only the protocol level will know how to construct the reply.

Parameters

- **context** (*any*) – A context returned by `receive_message()`.
- **reply** (*bytes*) – The reply to return to the client.

classmethod create(*host: str*, *queue: str*, *exchange: str = ''*) → tinyrpc.transports.rabbitmq.RabbitMQServerTransport
Create new server transport.

Instead of creating the BlockingConnection yourself, you can call this function and pass in the host name, queue, and exchange.

Parameters

- **host** – The host clients will connect to.
- **queue** – The RabbitMQ queue to consume messages from.
- **exchange** – The RabbitMQ exchange to use.

class tinyrpc.transports.rabbitmq.**RabbitMQClientTransport**(*connection: pika.adapters.blocking_connection.BlockingCon routing_key: str*, *exchange: str = ''*)

Bases: `tinyrpc.transports.ClientTransport`

Client transport based on a `pika.BlockingConnection`.

The transport assumes a RabbitMQ topology has already been established.

Parameters

- **connection** – A `pika.BlockingConnection` instance.
- **routing_key** – The RabbitMQ routing key to direct messages.

- **exchange** – The RabbitMQ exchange to use.

send_message(*message: bytes*, *expect_reply: bool = True*) → bytes

Send a message to the server and possibly receive a reply.

Sends a message to the connected server.

The message must be treated as a binary entity as only the protocol level will know how to interpret the message.

If the transport encodes the message in some way, the opposite end is responsible for decoding it before it is passed to either client or server.

This function will block until the reply has been received.

Parameters

- **message** (*bytes*) – The request to send to the server.
- **expect_reply** (*bool*) – Some protocols allow notifications for which a reply is not expected. When this flag is `False` the transport may not wait for a response from the server. **Note** that it is still the responsibility of the transport layer how to implement this. It is still possible that the server sends some form of reply regardless the value of this flag.

Returns The servers reply to the request.

Return type bytes

classmethod create(*host: str*, *routing_key: str*, *exchange: str = ''*) → tinyrpc.transports.rabbitmq.RabbitMQClientTransport

Create new client transport.

Instead of creating the BlockingConnection yourself, you can call this function and pass in the host name, routing key, and exchange.

Parameters

- **host** – The host clients will connect to.
- **routing_key** – The RabbitMQ routing key to direct messages.
- **exchange** – The RabbitMQ exchange to use.

WebSocket

class tinyrpc.transports.websocket.**WSServerTransport**(*queue_class: queue.Queue = <class 'queue.Queue'>*, *wsgi_handler: Callable[[], str] = None*)

Bases: `tinyrpc.transports.ServerTransport`

Requires `geventwebsocket`.

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

This is due to `handle` which is a `geventwebsocket.resource.Resource` that joins a wsgi handler for the / and a WebSocket handler for the /ws path. These resource is used in combination with a `geventwebsocket.server.WebSocketServer` that blocks 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

- **queue_class** – The queue class to use.
- **wsgi_handler** – Can be used to change the standard response to a http request to the /

receive_message() → Tuple[Any, bytes]
Receive a message from the transport.

Blocks until a message has been received. May return an opaque context object to its caller that should be passed on to `send_reply()` to identify the transport or requester later on. Use and function of the context object are entirely controlled by the transport instance.

The message must be treated as a binary entity as only the protocol level will know how to interpret the message.

If the transport encodes the message in some way, the opposite end is responsible for decoding it before it is passed to either client or server.

Returns A tuple consisting of `(context, message)`. Where `context` can be any valid Python type and `message` must be a `bytes` object.

send_reply(*context: Any*, *reply: bytes*) → None
Sends a reply to a client.

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

The reply must be a bytes object since only the protocol level will know how to construct the reply.

Parameters

- **context** (*any*) – A context returned by `receive_message()`.
- **reply** (*bytes*) – The reply to return to the client.

class tinyrpc.transports.websocket.**WSApplication**(*ws*)
Bases: geventwebsocket.resource.WebSocketApplication

This class is the bridge between the WSServerTransport and the WebSocket protocol implemented by geventwebsocket.resource.WebSocketApplication

class tinyrpc.transports.websocketclient.**HttpWebSocketClientTransport**(*endpoint: str*, ***kwargs*)
Bases: tinyrpc.transports.ClientTransport

HTTP WebSocket based client transport.

Requires `websocket-python`. Submits messages to a server using the body of an `HTTP WebSocket` message. Replies are taken from the response of the websocket.

The connection is establish on the `__init__` because the protocol is connection oriented, you need to close the connection calling the close method.

Parameters

- **endpoint** – The URL to connect the websocket.
- **kwargs** – Additional parameters for `websocket.send()`.

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

Sends a message to the connected server.

The message must be treated as a binary entity as only the protocol level will know how to interpret the message.

If the transport encodes the message in some way, the opposite end is responsible for decoding it before it is passed to either client or server.

This function will block until the reply has been received.

Parameters

- **message** (*bytes*) – The request to send to the server.
- **expect_reply** (*bool*) – Some protocols allow notifications for which a reply is not expected. When this flag is `False` the transport may not wait for a response from the server. **Note** that it is still the responsibility of the transport layer how to implement this. It is still possible that the server sends some form of reply regardless the value of this flag.

Returns The servers reply to the request.

Return type bytes

close() → None
Terminate the connection.

Since WebSocket maintains an open connection over multiple calls it must be closed explicitly.

1.8 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: tinyrpc.protocols.RPCProtocol*, *transport: tinyrpc.transports.ClientTransport*)

Bases: `object`

Client for making RPC calls to connected servers.

Parameters

- **protocol** (`RPCProtocol`) – An `RPCProtocol` instance.
- **transport** (*ClientTransport*) – The data transport mechanism

batch_call(*calls: List[tinyrpc.client.RPCCallTo]*) → tinyrpc.protocols.RPCBatchResponse
Experimental, use at your own peril.

call(*method: str, args: List[T], kwargs: Dict[KT, VT], one_way: bool = False*) → Any
Calls the requested method and returns the result.

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

Parameters

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

Returns The result of the call

Return type any

call_all(*requests: List[tinyrpc.client.RPCCall]*) → List[Any]
Calls the methods in the request in parallel.

When the gevent module is already loaded it is assumed to be correctly initialized, including monkey patching if necessary. In that case the RPC calls defined by requests are performed in parallel otherwise the methods are called sequentially.

Parameters **requests** – A list of either RPCCall or RPCCallTo elements. When RPCCallTo is used each element defines a transport. Otherwise the default transport set when RPCClient is created is used.

Returns A list with replies matching the order of the requests.

get_proxy(*prefix: str = ''*, *one_way: bool = False*) → tinyrpc.client.RPCProxy
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: tinyrpc.client.RPCClient*, *prefix: str = ''*, *one_way: bool = False*)
Bases: object

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().

class tinyrpc.client.**RPCCall**(*method*, *args*, *kwargs*)
Bases: tuple

Defines the elements of an RPC call.

RPCCall is used with call_all() to provide the list of requests to be processed. Each request contains the elements defined in this tuple.

args
Alias for field number 1

kwargs
Alias for field number 2

method
Alias for field number 0

class tinyrpc.client.**RPCCallTo**(*transport*, *method*, *args*, *kwargs*)
Bases: tuple

Defines the elements of a RPC call directed to multiple transports.

RPCCallTo is used with call_all() to provide the list of requests to be processed.

args
Alias for field number 2

kwargs
Alias for field number 3

method
Alias for field number 1

transport
Alias for field number 0

1.9 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.

Server definition.

Defines and implements a single-threaded, single-process, synchronous server.

class tinyrpc.server.**RPCServer**(*transport: tinyrpc.transports.ServerTransport*, *protocol: tinyrpc.protocols.RPCProtocol*, *dispatcher: tinyrpc.dispatch.RPCDispatcher*)

High level RPC server.

The server is completely generic only assuming some form of RPC communication is intended. Protocol, data transport and method dispatching are injected into the server object.

Parameters

- **transport** (ServerTransport) – The data transport mechanism to use.
- **protocol** (*RPCProtocol*) – The RPC protocol to use.
- **dispatcher** (*RPCDispatcher*) – The dispatching mechanism to use.

receive_one_message() → None
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 resulting request and the return value (either an error or a result) will be sent back to the client using the transport.

serve_forever() → None
Handle requests forever.

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

trace = None
Trace incoming and outgoing messages.

When this attribute is set to a callable this callable will be called directly after a message has been received and immediately after a reply is sent. The callable should accept three positional parameters:

Parameters

- **direction** (*str*) – Either '–>' for incoming or '<–' for outgoing data.
- **context** (*any*) – The context returned by `receive_message()`.
- **message** (*bytes*) – The message itself.

Example:

```
def my_trace(direction, context, message):
    logger.debug('%s%s', direction, message)

server = RPCServer(transport, protocol, dispatcher)
server.trace = my_trace
server.serve_forever()
```

will log all incoming and outgoing traffic of the RPC service.

Note that the `message` will be the data stream that is transported, not the interpreted meaning of that data. It is therefore possible that the binary stream is unreadable without further translation.

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.

1.10 The Exceptions hierarchy

All exceptions are rooted in the `Exception` class. The *RPCError* class derives from it and forms the basis of all tinyrpc exceptions.

1.10.1 Abstract exceptions

These exceptions, most of them will be overridden, define errors concerning the transport and structure of messages.

class tinyrpc.exc.**RPCError**

Bases: `Exception`, `abc.ABC`

Base class for all exceptions thrown by `tinyrpc`.

error_respond()

Converts the error to an error response object.

Returns An error response instance or `None` if the protocol decides to drop the error silently.

Return type *RPCErrorResponse*

class tinyrpc.exc.**BadRequestError**

Bases: *tinyrpc.exc.RPCError*, `abc.ABC`

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

class tinyrpc.exc.**BadReplyError**

Bases: *tinyrpc.exc.RPCError*, `abc.ABC`

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

class tinyrpc.exc.**InvalidRequestError**
Bases: *tinyrpc.exc.BadRequestError*, abc.ABC

A request made was malformed (i.e. violated the specification) and could not be parsed.

class tinyrpc.exc.**InvalidReplyError**
Bases: *tinyrpc.exc.BadReplyError*, abc.ABC

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

class tinyrpc.exc.**MethodNotFoundError**
Bases: *tinyrpc.exc.RPCError*, abc.ABC

The desired method was not found.

class tinyrpc.exc.**InvalidParamsError**
Bases: *tinyrpc.exc.RPCError*, abc.ABC

The provided parameters do not match those of the desired method.

class tinyrpc.exc.**ServerError**
Bases: *tinyrpc.exc.RPCError*, abc.ABC

An internal error in the RPC system occurred.

1.10.2 Protocol exceptions

Each protocol provides its own concrete implementations of these exceptions.

JSON-RPC

class tinyrpc.protocols.jsonrpc.**JSONRPCParseError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.jsonrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

The request cannot be decoded or is malformed.

class tinyrpc.protocols.jsonrpc.**JSONRPCInvalidRequestError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.jsonrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

The request contents are not valid for JSON RPC 2.0

class tinyrpc.protocols.jsonrpc.**JSONRPCMethodNotFoundError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.jsonrpc.FixedErrorMessageMixin*, *tinyrpc.exc.MethodNotFoundError*

The requested method name is not found in the registry.

class tinyrpc.protocols.jsonrpc.**JSONRPCInvalidParamsError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.jsonrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

The provided parameters are not appropriate for the function called.

class tinyrpc.protocols.jsonrpc.**JSONRPCInternalError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.jsonrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

Unspecified error, not in the called function.

class tinyrpc.protocols.jsonrpc.**JSONRPCServerError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.jsonrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

Unspecified error, this message originates from the called function.

This last exception is a client side exception designed to represent the server side error in the client.

class tinyrpc.protocols.jsonrpc.**JSONRPCError**(*error: Union[JSONRPCErrorResponse, Dict[str, Any]]*)
Bases: *tinyrpc.protocols.jsonrpc.FixedErrorMessageMixin*, *tinyrpc.exc.RPCError*

Reconstructs (to some extend) the server-side exception.

The client creates this exception by providing it with the `error` attribute of the JSON error response object returned by the server.

Parameters **error** (*dict*) – This dict contains the error specification:

- code (int): the numeric error code.
- message (str): the error description.
- data (any): if present, the data attribute of the error

MSGPACK-RPC

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCParseError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.msgpackrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCInvalidRequestError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.msgpackrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCMethodNotFoundError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.msgpackrpc.FixedErrorMessageMixin*, *tinyrpc.exc.MethodNotFoundError*

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCInvalidParamsError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.msgpackrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCInternalError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.msgpackrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCServerError**(**args*, ***kwargs*)
Bases: *tinyrpc.protocols.msgpackrpc.FixedErrorMessageMixin*, *tinyrpc.exc.InvalidRequestError*

This last exception is a client side exception designed to represent the server side error in the client.

class tinyrpc.protocols.msgpackrpc.**MSGPACKRPCError**(*error: Union[MSGPACKRPCErrorResponse, Tuple[int, str]]*)
Bases: *tinyrpc.protocols.msgpackrpc.FixedErrorMessageMixin*, *tinyrpc.exc.RPCError*

Reconstructs (to some extend) the server-side exception.

The client creates this exception by providing it with the `error` attribute of the MSGPACK error response object returned by the server.

Parameters **`error`** – This tuple contains the error specification: the numeric error code and the error description.

CHAPTER 2

Installation

```
pip install tinyrpc
```

will install `tinyrpc` with its default dependencies.

2.1 Optional dependencies

Depending on the protocols and transports you want to use additional dependencies are required. You can instruct pip to install these dependencies by specifying extras to the basic install command.

```
pip install tinyrpc[httpclient, wsgi]
```

will install `tinyrpc` with dependencies for the httpclient and wsgi transports.

Available extras are:

Option	Needed to use objects of class
gevent	optional in RPCClient, required by RPCServerGreenlets
httpclient	HttpPostClientTransport, HttpWebSocketClientTransport
jsonext	optional in JSONRPCProtocol
msgpack	required by MSGPACKRPCProtocol
rabbitmq	RabbitMQServerTransport, RabbitMQClientTransport
websocket	WSServerTransport, HttpWebSocketClientTransport
wsgi	WsgiServerTransport
zmq	ZmqServerTransport, ZmqClientTransport

CHAPTER 3

People

3.1 Creator

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

  As of this writing (in Jan 2013) there are a few jsonrpc libraries already out there on PyPI, most of them handling one specific use case (e.g. json via WSGI, using Twisted, or TCP-sockets).

  None of the libraries, however, made it easy to reuse the jsonrpc-parsing bits and substitute a different transport (i.e. going from json via TCP to an implementation using WebSockets or 0mq).

  In the end, all these libraries have their own dispatching interfaces and a custom implementation of handling jsonrpc.

  `tinyrpc` aims to do better by dividing the problem into cleanly interchangeable parts that allow easy addition of new transport methods, RPC protocols or dispatchers.

3.2 Maintainer

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

  Looking for a Python jsonrpc library I found `tinyrpc`. I was immediately taken by its modular concept and construction.

  After creating a couple transports and trying to get them integrated in tinyrpc, I learned that Marc got involved with other projects and that maintaining `tinyrpc` became too much a burden. I then volunteered to become its maintainer.

Index

Symbols

A

B

C

D

E

F

G

I

S

T

U

V