
TChannel Documentation

Release 0.1.0

Uber Technologies, Inc.

September 22, 2015

1	Getting Started	3
1.1	Initial Setup	3
1.2	Thrift Interface Definition	3
1.3	Python Server	4
1.4	Handlers	5
1.5	Hyperbahn	6
1.6	Debugging	6
1.7	Python Client	7
2	API Documentation	9
2.1	TChannel	9
2.2	Exceptions	13
2.3	Thrift	13
2.4	Synchronous Client	14
2.5	Testing	16
3	Changelog	19
3.1	0.14.0 (2015-08-03)	19
3.2	0.13.0 (2015-07-23)	19
3.3	0.12.0 (2015-07-20)	19
3.4	0.11.2 (2015-07-20)	19
3.5	0.11.1 (2015-07-17)	20
3.6	0.11.0 (2015-07-17)	20
3.7	0.10.3 (2015-07-13)	20
3.8	0.10.2 (2015-07-13)	20
3.9	0.10.1 (2015-07-10)	20
3.10	0.10.0 (2015-07-10)	20
3.11	0.9.1 (2015-07-09)	20
3.12	0.9.0 (2015-07-07)	20
3.13	0.8.5 (2015-06-30)	21
3.14	0.8.4 (2015-06-17)	21
3.15	0.8.3 (2015-06-15)	21
3.16	0.8.2 (2015-06-11)	21
3.17	0.8.1 (2015-06-10)	21
	Python Module Index	23

A Python implementation of [TChannel](#).

Getting Started

The code matching this guide is [here](#).

1.1 Initial Setup

Create a directory called `keyvalue` to work inside of:

```
$ mkdir ~/keyvalue
$ cd ~/keyvalue
```

Inside of this directory we're also going to create a `keyvalue` module, which requires an `__init__.py` and a `setup.py` at the root:

```
$ mkdir keyvalue
$ touch keyvalue/__init__.py
```

Setup a [virtual environment](#) for your service and install the `tornado` and `tchannel`:

```
$ virtualenv env
$ source env/bin/activate
$ pip install tchannel
```

1.2 Thrift Interface Definition

Create a [Thrift](#) file under `thrift/service.thrift` that defines an interface for your service:

```
$ cat thrift/service.thrift
```

```
exception NotFoundError {
    1: string key,
}

service KeyValue {
    string getValue(
        1: string key,
    ) throws (
        1: NotFoundError notFound,
    )

    void setValue(
```

```
    1: string key,
    2: string value,
)
}
```

This defines a service named `KeyValue` with two functions:

getValue a function which takes one string parameter, and returns a string.

setValue a void function that takes in two parameters.

Once you have defined your service, generate corresponding Thrift types by running the following:

```
$ thrift --gen py:new_style,dynamic,slots,utf8strings \
    -out keyvalue thrift/service.thrift
```

This generates client- and server-side code to interact with your service.

You may want to verify that your thrift code was generated successfully:

```
$ python -m keyvalue.service.KeyValue
```

1.3 Python Server

To serve an application we need to instantiate a `TChannel` instance, which we will register handlers against. Open up `keyvalue/server.py` and write something like this:

```
from __future__ import absolute_import

from tornado import ioloop

from service import KeyValue
from tchannel.tornado import TChannel

app = TChannel('keyvalue-server')

@app.register(KeyValue)
def getValue(request, response, tchannel):
    pass

@app.register(KeyValue)
def setValue(request, response, tchannel):
    pass

def run():
    app.listen()
    ioloop.IOLoop.current().start()

if __name__ == '__main__':
    run()
```

Here we have created a `TChannel` instance and registered two no-op handlers with it. The name of these handlers map directly to the Thrift service we defined earlier.

NOTE: Method handlers do not need to be declared at import-time, since this can become unwieldy in complex applications. We could also define them like so:

```
def run():
    app = TChannel('keyvalue-server')
    app.register(KeyValue, handler=Get)
    app.register(KeyValue, handler=Set)
    app.listen()
    ioloop.IOLoop.current().start()
```

A TChannel server only has one requirement: a name for itself. By default an ephemeral port will be chosen to listen on (although an explicit port can be provided).

(As your application becomes more complex, you won't want to put everything in a single file like this. Good code structure is beyond the scope of this guide.)

Let's make sure this server is in a working state:

```
python keyvalue/server.py
^C
```

The process should hang until you kill it, since it's listening for requests to handle. You shouldn't get any exceptions.

1.4 Handlers

To implement our service's endpoints let's create an in-memory dictionary that our endpoints will manipulate:

```
values = {}

@app.register(KeyValue)
def getValue(request, response, tchannel):
    key = request.args.key
    value = values.get(key)

    if value is None:
        raise KeyValue.NotFoundError(key)

    return value

@app.register(KeyValue)
def setValue(request, response, tchannel):
    key = request.args.key
    value = request.args.value
    values[key] = value
```

You can see that the return value of `Get` will be coerced into the expected Thrift shape. If we needed to return an additional field, we could accomplish this by returning a dictionary.

This example service doesn't do any network IO work. If we wanted to take advantage of Tornado's [asynchronous](#) capabilities, we could define our handlers as coroutines and yield to IO operations:

```
@app.register(KeyValue)
@tornado.gen.coroutine
def setValue(request, response, tchannel):
    key = request.args.key
    value = request.args.value
```

```
# Simulate some non-blocking IO work.
yield tornado.gen.sleep(1.0)

values[key] = value
```

You have probably noticed that all of these handlers are passed `response` and `tchannel` objects, in addition to a `request`. The `response` object is available for advanced use cases where it doesn't make sense to return one object as a response body – for example, long-lived connections that gradually stream the response back to the caller.

The `tchannel` object contains context about the current request (such as Zipkin tracing information) and should be used to make requests to other TChannel services. (Note that this API may change in the future.)

1.4.1 Transport Headers

In addition to the call arguments and headers, the `request` object also provides some additional information about the current request under the `request.transport` object:

`transport.flags` Request flags used by the protocol for fragmentation and streaming.

`transport.ttl` The time (in milliseconds) within which the caller expects a response.

`transport.headers` Protocol level headers for the request. For more information on transport headers check the [Transport Headers](#) section of the protocol document.

1.5 Hyperbahn

As mentioned earlier, our service is listening on an ephemeral port, so we are going to register it with the Hyperbahn routing mesh. Clients will use this Hyperbahn mesh to determine how to communicate with your service.

Let's change our `run` method to advertise our service with a local Hyperbahn instance:

```
def run():
    app.listen()
    app.advertise(['localhost:23000'], 'keyvalue-server')
    ioloop.IOLoop.current().start()
```

The `advertise` method takes a seed list of Hyperbahn routers and the name of the service that clients will call into. After advertising, the Hyperbahn will connect to your process and establish peers for service-to-service communication.

Consult the Hyperbahn documentation for instructions on how to start a process locally.

1.6 Debugging

Let's spin up the service and make a request to it through Hyperbahn. Python provides `tcurl.py` script, but we need to use the [Node version](#) for now since it has Thrift support.

```
$ python keyvalue/server.py &
$ node tcurl -p localhost:23000 -t ~/keyvalue/thrift service KeyValue::Set -3 '{"key": "hello", "value": "world"}'
$ node tcurl -p localhost:23000 -t ~/keyvalue/thrift service KeyValue::Get -3 '{"key": "hello"}'
$ node tcurl -p localhost:23000 -t ~/keyvalue/thrift service KeyValue::Get -3 '{"key": "hi"}'
```

Your service can now be accessed from any language over Hyperbahn + TChannel!

1.7 Python Client

Let's make a client call from Python in `keyvalue/client.py`:

```
from tornado import gen
from tornado import ioloop
from tchannel.thrift import client_for

from service import KeyValue

KeyValueClient = client_for('keyvalue-server', KeyValue)

@gen.coroutine
def run():
    app_name = 'keyvalue-client'

    app = TChannel(app_name)
    app.advertise(['localhost:23000'], app_name)

    client = KeyValueClient(app)

    yield client.Set("foo", "bar")

    response = yield client.Get("foo")

    print response

if __name__ == '__main__':
    ioloop.IOLoop.current().run_sync(run)
```

Similar to the server case, we initialize a `TChannel` instance and advertise ourselves on Hyperbahn (to establish how to communicate with *keyval-server*). After this we create a client class to add `TChannel` functionality to our generated Thrift code. We then set and retrieve a value from our server.

API Documentation

2.1 TChannel

class `tchannel.tornado.TChannel` (*name*, *hostport=None*, *process_name=None*, *known_peers=None*, *trace=False*)

Manages inbound and outbound connections to various hosts.

advertise (*routers*, *name=None*, *timeout=None*)

Make a service available on the Hyperbahn routing mesh.

This will make contact with a Hyperbahn host from a list of known Hyperbahn routers. Additional Hyperbahn connections will be established once contact has been made with the network.

Parameters

- **router** – A seed list of addresses of Hyperbahn routers, e.g., `["127.0.0.1:23000"]`.
- **name** – The identity of this service on the Hyperbahn.

This is usually unnecessary, as it defaults to the name given when initializing the `TChannel` (which is used as your identity as a caller).

Returns

A future that resolves to the remote server's response after the first advertise finishes.

Advertisement will continue to happen periodically.

listen (*port=None*)

Start listening for incoming connections.

A request handler must have already been specified with `TChannel.host`.

Parameters **port** – An explicit port to listen on. This is unnecessary when advertising on Hyperbahn.

Returns Returns immediately.

Raises **AlreadyListeningError** If listen was already called.

register (*endpoint*, *scheme=None*, *handler=None*, ***kwargs*)

Register a handler with this `TChannel`.

This may be used as a decorator:

```
app = TChannel(name='foo')

@app.register("hello", "json")
def hello_handler(request, response, tchannel):
    params = yield request.get_body()
```

Or as a function:

```
# Here we have a Thrift handler for `Foo::hello`
app.register(Foo, "hello", hello_thrift_handler)
```

Parameters

- **endpoint** – Name of the endpoint being registered. This should be a reference to the Thrift-generated module if this is a Thrift endpoint. It may also be `TChannel.FALLBACK` if it's intended to be a catch-all endpoint.
- **scheme** – Name of the scheme under which the endpoint is being registered. One of “raw”, “json”, and “thrift”. Defaults to “raw”, except if “endpoint” was a module, in which case this defaults to “thrift”.
- **handler** – If specified, this is the handler function. If ignored, this function returns a decorator that can be used to register the handler function.

Returns If `handler` was specified, this returns `handler`. Otherwise, it returns a decorator that can be applied to a function to register it as the handler.

request (*hostport=None, service=None, arg_scheme=None, retry=None, **kwargs*)

Initiate a new request through this TChannel.

Parameters

- **hostport** – Host to which the request will be made. If unspecified, a random known peer will be picked. This is not necessary if using Hyperbahn.
- **service** – The name of a service available on Hyperbahn. Defaults to an empty string.
- **arg_scheme** – Determines the serialization scheme for the request. One of ‘raw’, ‘json’, or ‘thrift’. Defaults to ‘raw’.
- **retry** – One of ‘n’ (never retry), ‘c’ (retry on connection errors), ‘t’ (retry on timeout), ‘ct’ (retry on connection errors and timeouts).

Defaults to ‘c’.

class `tchannel.tornado.RequestDispatcher`

A synchronous RequestHandler that dispatches calls to different endpoints based on `arg1`.

Endpoints are registered using `register` or the `route` decorator.

```
handler = # ...

@handler.route('my_method')
def my_method(request, response, proxy):
    response.write('hello world')
```

static not_found (*request, response, proxy*)

Default behavior for requests to unrecognized endpoints.

register (*rule, handler, broker=None*)

Register a new endpoint with the given name.

```
@dispatcher.register('is_healthy')
def check_health(request, response, proxy):
    # ...
```

Parameters

- **rule** – Name of the endpoint. Incoming Call Requests must have this as `arg1` to dispatch to this handler.

If `RequestHandler.FALLBACK` is specified as a rule, the given handler will be used as the ‘fallback’ handler when requests don’t match any registered rules.

- **handler** – A function that gets called with `Request`, `Response`, and the `proxy`.
 - **broker** – Broker injects customized serializer and deserializer into request/response object.
- `broker==None` means it registers as raw handle. It deals with raw buffer in the request/response.

route (*rule*, *helper=None*)

See `register` for documentation.

class `tchannel.tornado.Request` (*id=None*, *flags=0*, *ttl=1000*, *tracing=None*, *service=None*, *headers=None*, *checksum=None*, *argstreams=None*, *scheme=None*, *endpoint=None*)

Represents an incoming request to an endpoint.

`Request` class is used to represent the `CallRequestMessage` at User’s level. This is going to hide the protocol level message information.

get_body (**args*, ***kwargs*)

Get the body value from the request.

Returns a future contains the deserialized value of body

get_body_s ()

Get the raw stream of body.

Returns the argstream of body

get_header (**args*, ***kwargs*)

Get the header value from the request.

Returns a future contains the deserialized value of header

get_header_s ()

Get the raw stream of header.

Returns the argstream of header

should_retry_on_error (*error*)

rules for retry

Parameters **error** – `ProtocolException` that returns from Server

class `tchannel.tornado.Response` (*connection=None*, *id=None*, *flags=None*, *code=None*, *tracing=None*, *headers=None*, *checksum=None*, *argstreams=None*, *scheme=None*)

An outgoing response.

`Response` class is used to represent the `CallResponseMessage` at User’s level. This is going to hide the protocol level message information.

flush()

Flush the response buffer.

No more write or set operations is allowed after flush call.

get_body(*args, **kwargs)

Get the body value from the response.

Returns a future contains the deserialized value of body

get_body_s()

Get the raw stream of body.

Returns the argstream of body

get_header(*args, **kwargs)

Get the header value from the response.

Returns a future contains the deserialized value of header

get_header_s()

Get the raw stream of header.

Returns the argstream of header

set_body_s(stream)

Set customized body stream.

Note: the body stream can only be changed before the stream is consumed.

Parameters stream – InMemStream/PipeStream for body

Raises TChannelError Raise TChannelError if the stream is being sent when you try to change the stream.

set_header_s(stream)

Set customized header stream.

Note: the header stream can only be changed before the stream is consumed.

Parameters stream – InMemStream/PipeStream for header

Raises TChannelError Raise TChannelError if the stream is being sent when you try to change the stream.

write_body(chunk)

Write to header.

Note: whenever write_body is called, the header stream will be closed. write_header method is unavailable.

Parameters chunk – content to write to body

Raises TChannelError Raise TChannelError if the response's flush() has been called

write_header(chunk)

Write to header.

Note: the header stream is only available to write before write body.

Parameters chunk – content to write to header

Raises TChannelError Raise TChannelError if the response's flush() has been called

2.2 Exceptions

exception `tchannel.errors.AdvertiseError`

Represent advertise failure exception

exception `tchannel.errors.AlreadyListeningError`

Represents exception from attempting to listen multiple times.

exception `tchannel.errors.InvalidChecksumError`

Represent invalid checksum type in the message

exception `tchannel.errors.InvalidEndpointError`

Represent an message containing invalid endpoint.

exception `tchannel.errors.InvalidErrorCodeError` (*code*)

Represent Invalid Error Code exception

exception `tchannel.errors.InvalidMessageError`

Represent an invalid message.

exception `tchannel.errors.NoAvailablePeerError`

Represents a failure to find any peers for a request.

exception `tchannel.errors.ProtocolError` (*code, description, id=None, tracing=None*)

Represent a protocol-level exception

exception `tchannel.errors.ReadError`

Raised when there is an error while reading input.

exception `tchannel.errors.StreamingError`

Represent Streaming Message Exception

exception `tchannel.errors.TChannelApplicationError` (*code, args*)

The remote application returned an exception.

This is not a protocol error. This means a response was received with the `code` flag set to fail.

exception `tchannel.errors.TChannelError`

Represent a TChannel-generated exception.

2.3 Thrift

`tchannel.thrift.client.client_for` (*service, service_module, thrift_service_name=None*)

Build a client class for the given Thrift service.

The generated class accepts a TChannel and an optional hostport as initialization arguments.

Given `CommentService` defined in `comment.thrift` and registered with Hyperbahn under the name “comment”, here’s how this may be used:

```
from comment import CommentService

CommentServiceClient = client_for("comment", CommentService)

@gen.coroutine
def post_comment(articleId, msg, hostport=None):
    client = CommentServiceClient(tchannel, hostport)
    yield client.postComment(articleId, CommentService.Comment(msg))
```

Parameters

- **service** – Name of the Hyperbahn service being called. This is the name with which the service registered with Hyperbahn.
- **service_module** – The Thrift-generated module for that service. This usually has the same name as defined for the service in the IDL.
- **thrift_service_name** – If the Thrift service has a different name than its module, use this parameter to specify it.

Returns An object with the same interface as the service that uses the given TChannel to call the service.

```
tchannel.thrift.client.generate_method(service_module, service_name, method_name)
```

Generate a method for the given Thrift service.

Parameters

- **service_module** – Thrift-generated service module
- **service_name** – Name of the Thrift service
- **method_name** – Method being called

2.4 Synchronous Client

```
class tchannel.sync.client.Response(header, body)
```

body

Alias for field number 1

header

Alias for field number 0

```
class tchannel.sync.client.SyncClientOperation(operation, threadloop)
```

Allows making client operation requests synchronously.

This object acts like `tchannel.TChannelClientOperation`, but instead uses a threadloop to make the request synchronously.

```
send(arg1, arg2, arg3)
```

Send the given triple over the wire.

Parameters

- **arg1** – String containing the contents of arg1. If None, an empty string is used.
- **arg2** – String containing the contents of arg2. If None, an empty string is used.
- **arg3** – String containing the contents of arg3. If None, an empty string is used.

Return concurrent.futures.Future Future response from the peer.

```
class tchannel.sync.client.TChannelSyncClient(name, process_name=None,
                                             known_peers=None, trace=False)
```

Make synchronous TChannel requests.

This client does not support incoming connections or requests- this is a uni-directional client only.

The client is implemented on top of the Tornado-based implementation and starts and stops IOLoops on-demand.

```

client = TChannelSyncClient()
response = client.request(
    hostport='localhost:4040',
    service='HelloService',
).send(
    'hello', None, json.dumps({"name": "World"})
)

```

advertise (*routers, name=None, timeout=None*)

Advertise with Hyperbahn.

Parameters

- **routers** – list of hyperbahn addresses to advertise to.
- **name** – service name to advertise with.
- **timeout** – backoff period for failed requests.

Returns first advertise result.

Raises `AdvertiseError` when unable to begin advertising.

request (**args, **kwargs*)

Initiate a new request to a peer.

Parameters

- **hostport** – If specified, requests will be sent to the specific host. Otherwise, a known peer will be picked at random.
- **service** – Name of the service being called. Defaults to an empty string.
- **service_threshold** – If `hostport` was not specified, this specifies the score threshold at or below which peers will be ignored.

Returns `SyncClientOperation` An object with a `send(arg1, arg2, arg3)` operation.

`tchannel.sync.thrift.client_for(service, service_module, thrift_service_name=None)`

Build a synchronous client class for the given Thrift service.

The generated class accepts a `TChannelSyncClient` and an optional `hostport` as initialization arguments.

Given `CommentService` defined in `comment.thrift` and registered with Hyperbahn under the name “comment”, here’s how this might be used:

```

from tchannel.sync import TChannelSyncClient
from tchannel.sync.thrift import client_for

from comment import CommentService

CommentServiceClient = client_for('comment', CommentService)

tchannel_sync = TChannelSyncClient('my-service')
comment_client = CommentServiceClient(tchannel_sync)

future = comment_client.postComment(
    articleId,
    CommentService.Comment("hi")
)
result = future.result()

```

Parameters

- **service** – Name of the Hyperbahn service being called.
- **service_module** – The Thrift-generated module for that service. This usually has the same name as defined for the service in the IDL.
- **thrift_service_name** – If the Thrift service has a different name than its module, use this parameter to specify it.

Returns An Thrift-like class, ready to be instantiated and used with TChannelSyncClient.

`tchannel.sync.thrift.generate_method(method_name)`

Generate a method for a given Thrift service.

Uses the provided TChannelSyncClient's threadloop in order to convert RPC calls to concurrent.futures

Parameters `method_name` – Method being called.

Returns A method that invokes the RPC using TChannelSyncClient

2.5 Testing

2.5.1 VCR

`tchannel.testing.vcr` provides VCR-like functionality for TChannel. Its API is heavily inspired by the [vcrpy](#) library.

This allows recording TChannel requests and their responses into YAML files during integration tests and replaying those recorded responses when the tests are run next time.

The simplest way to use this is with the `use_cassette()` function.

`tchannel.testing.vcr.use_cassette(path, record_mode=None, inject=False)`

Use or create a cassette to record/replay TChannel requests.

This may be used as a context manager or a decorator.

```
from tchannel.testing import vcr

@pytest.mark.gen_test
@vcr.use_cassette('tests/data/foo.yaml')
def test_foo():
    channel = TChannel('test-client')
    service_client = MyServiceClient(channel)

    yield service_client.myMethod()

def test_bar():
    with vcr.use_cassette('tests/data/bar.yaml', record_mode='none'):
        # ...
```

Note that when used as a decorator on a coroutine, the `use_cassette` decorator must be applied BEFORE `gen.coroutine` or `pytest.mark.gen_test`.

Parameters

- **path** – Path to the cassette. If the cassette did not already exist, it will be created. If it existed, its contents will be replayed (depending on the record mode).

- **record_mode** – The record mode dictates whether a cassette is allowed to record or replay interactions. This may be a string specifying the record mode name or an element from the `tchannel.testing.vcr.RecordMode` object. This parameter defaults to `tchannel.testing.vcr.RecordMode.ONCE`. See `tchannel.testing.vcr.RecordMode` for details on supported record modes and how to use them.
- **inject** – If True, when `use_cassette` is used as a decorator, the cassette object will be injected into the function call as the first argument. Defaults to False.

Configuration

Record Modes

class `tchannel.testing.vcr.RecordMode`

Record modes dictate how a cassette behaves when interactions are replayed or recorded. The following record modes are supported.

ONCE = 'once'

If the YAML file did not exist, record new interactions and save them. If the YAML file already existed, replay existing interactions but disallow any new interactions. This is the default and usually what you want.

NEW_EPISODES = 'new_episodes'

Replay existing interactions and allow recording new ones. This is usually undesirable since it reduces predictability in tests.

NONE = 'none'

Replay existing interactions and disallow any new interactions. This is a good choice for tests whose behavior is unlikely to change in the near future. It ensures that those tests don't accidentally start making new requests.

ALL = 'all'

Do not replay anything and record all new interactions. Forget all existing interactions. This may be used to record everything anew.

Changelog

3.1 0.14.0 (2015-08-03)

- Implement VCR functionality for outgoing requests. Check the documentation for `tchannel.testing.vcr` for details.
- Add support for specifying fallback handlers via `TChannel.register` by specifying `TChannel.fallback` as the endpoint.
- Fix bug in `Response` where `code` expected an object instead of an integer.
- Fix bug in `Peer.close` where a future was expected instead of `None`.

3.2 0.13.0 (2015-07-23)

- Add support for specifying transport headers for Thrift clients.
- Always pass `shardKey` for `TCollector` tracing calls. This fixes Zipkin tracing for Thrift clients.

3.3 0.12.0 (2015-07-20)

- Add `TChannel.is_listening()` to determine if `listen` has been called.
- Calling `TChannel.listen()` more than once raises a `tchannel.errors.AlreadyListeningError`.
- `TChannel.advertise()` will now automatically start listening for connections if `listen()` has not already been called.
- Use `threadloop==0.4`.
- Removed `print_arg`.

3.4 0.11.2 (2015-07-20)

- Fix sync client's `advertise` - needed to call `listen` in thread.

3.5 0.11.1 (2015-07-17)

- Fix sync client using `0.0.0.0` host which gets rejected by Hyperbahn during advertise.

3.6 0.11.0 (2015-07-17)

- Added advertise support to sync client in `tchannel.sync.TChannelSyncClient.advertise`.
- **BREAKING** - renamed `router` argument to `routers` in `tchannel.tornado.TChannel.advertise`.

3.7 0.10.3 (2015-07-13)

- Support PyPy 2.
- Fix bugs in `TChannel.advertise`.

3.8 0.10.2 (2015-07-13)

- Made `TChannel.advertise` retry on all exceptions.

3.9 0.10.1 (2015-07-10)

- Previous release was broken with older versions of pip.

3.10 0.10.0 (2015-07-10)

- Add exponential backoff to `TChannel.advertise`.
- Make transport metadata available under `request.transport` on the server-side.

3.11 0.9.1 (2015-07-09)

- Use threadloop 0.3.* to fix main thread not exiting when `tchannel.sync.TChannelSyncClient` is used.

3.12 0.9.0 (2015-07-07)

- Allow custom handlers for unrecognized endpoints.
- Released `tchannel.sync.TChannelSyncClient` and `tchannel.sync.thrift.client_for`.

3.13 0.8.5 (2015-06-30)

- Add port parameter for `TChannel.listen`.

3.14 0.8.4 (2015-06-17)

- Fix bug where `False` and `False`-like values were being treated as `None` in Thrift servers.

3.15 0.8.3 (2015-06-15)

- Add `as` attribute to the response header.

3.16 0.8.2 (2015-06-11)

- Fix callable `traceflag` being propagated to the serializer.
- Fix circular imports.
- Fix `TimeoutError` retry logic.

3.17 0.8.1 (2015-06-10)

- Initial release.

t

`tchannel.errors`, [13](#)
`tchannel.sync.client`, [14](#)
`tchannel.sync.thrift`, [15](#)
`tchannel.testing.vcr`, [16](#)
`tchannel.thrift.client`, [13](#)

A

advertise() (tchannel.sync.client.TChannelSyncClient method), 15
advertise() (tchannel.tornado.TChannel method), 9
AdvertiseError, 13
ALL (tchannel.testing.vcr.RecordMode attribute), 17
AlreadyListeningError, 13

B

body (tchannel.sync.client.Response attribute), 14

C

client_for() (in module tchannel.sync.thrift), 15
client_for() (in module tchannel.thrift.client), 13

F

flush() (tchannel.tornado.Response method), 11

G

generate_method() (in module tchannel.sync.thrift), 16
generate_method() (in module tchannel.thrift.client), 14
get_body() (tchannel.tornado.Request method), 11
get_body() (tchannel.tornado.Response method), 12
get_body_s() (tchannel.tornado.Request method), 11
get_body_s() (tchannel.tornado.Response method), 12
get_header() (tchannel.tornado.Request method), 11
get_header() (tchannel.tornado.Response method), 12
get_header_s() (tchannel.tornado.Request method), 11
get_header_s() (tchannel.tornado.Response method), 12

H

header (tchannel.sync.client.Response attribute), 14

I

InvalidChecksumError, 13
InvalidEndpointError, 13
InvalidErrorCodeError, 13
InvalidMessageError, 13

L

listen() (tchannel.tornado.TChannel method), 9

N

NEW_EPISODES (tchannel.testing.vcr.RecordMode attribute), 17
NoAvailablePeerError, 13
NONE (tchannel.testing.vcr.RecordMode attribute), 17
not_found() (tchannel.tornado.RequestDispatcher static method), 10

O

ONCE (tchannel.testing.vcr.RecordMode attribute), 17

P

ProtocolError, 13

R

ReadError, 13
RecordMode (class in tchannel.testing.vcr), 17
register() (tchannel.tornado.RequestDispatcher method), 10
register() (tchannel.tornado.TChannel method), 9
Request (class in tchannel.tornado), 11
request() (tchannel.sync.client.TChannelSyncClient method), 15
request() (tchannel.tornado.TChannel method), 10
RequestDispatcher (class in tchannel.tornado), 10
Response (class in tchannel.sync.client), 14
Response (class in tchannel.tornado), 11
route() (tchannel.tornado.RequestDispatcher method), 11

S

send() (tchannel.sync.client.SyncClientOperation method), 14
set_body_s() (tchannel.tornado.Response method), 12
set_header_s() (tchannel.tornado.Response method), 12
should_retry_on_error() (tchannel.tornado.Request method), 11
StreamingError, 13

[SyncClientOperation](#) (class in `tchannel.sync.client`), [14](#)

T

[TChannel](#) (class in `tchannel.tornado`), [9](#)

[tchannel.errors](#) (module), [13](#)

[tchannel.sync.client](#) (module), [14](#)

[tchannel.sync.thrift](#) (module), [15](#)

[tchannel.testing.vcr](#) (module), [16](#)

[tchannel.thrift.client](#) (module), [13](#)

[TChannelApplicationError](#), [13](#)

[TChannelError](#), [13](#)

[TChannelSyncClient](#) (class in `tchannel.sync.client`), [14](#)

U

[use_cassette\(\)](#) (in module `tchannel.testing.vcr`), [16](#)

W

[write_body\(\)](#) (`tchannel.tornado.Response` method), [12](#)

[write_header\(\)](#) (`tchannel.tornado.Response` method), [12](#)