TChannel Documentation

Release 0.1.0

Uber Technologies, Inc.

September 22, 2015

Contents

A Python implementation of TChannel.

CHAPTER 1

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 thrift tornado
```

1.2 Thrift Interface Definition

Create a Thrift file under `thrift/service.thrift` that defines an interface for your service:

```
$ mkdir thrift
$ vim thrift/service.thrift
$ 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:

```python
from __future__ import absolute_import

from tornado import ioloop
from tornado import gen

from service import KeyValue
from tchannel import TChannel


tchannel = TChannel('keyvalue-server')


@tchannel.thrift.register(KeyValue)
def getValue(request):
    pass


@tchannel.thrift.register(KeyValue)
def setValue(request):
    pass


def run():
    tchannel.listen()
    print('Listening on %s' % tchannel.hostport)


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

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.

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
Listening on localhost:54143
^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:

```python
values = {}


@tchannel.thrift.register(KeyValue)
def getValue(request):
    key = request.body.key
    value = values.get(key)

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

    return value


@tchannel.thrift.register(KeyValue)
def setValue(request):
    key = request.body.key
    value = request.body.value
    values[key] = value
```

You can see that the return value of `getValue` 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:

```python
@tchannel.register(KeyValue)
@gen.coroutine
def setValue(request):
    key = request.body.key
    value = request.body.value

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

    values[key] = value
```

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:

```python
import json
import os

@gen.coroutine
def run():

    tchannel.listen()
    print('Listening on %s' % app.hostport)

    if os.path.exists('/path/to/hyperbahn_hostlist.json'):
        with open('/path/to/hyperbahn_hostlist.json', 'r') as f:
            hyperbahn_hostlist = json.load(f)
        yield tchannel.advertise(routers=hyperbahn_hostlist)
```

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 &
$ tcurl -H /path/to/hyperbahn_host_list.json -t ~/keyvalue/thrift/service.thrift service KeyValue::se
$ tcurl -H /path/to/hyperbahn_host_list.json -t ~/keyvalue/thrift/service.thrift service KeyValue::ge
$ tcurl -H /path/to/hyperbahn_host_list.json -t ~/keyvalue/thrift/service.thrift service KeyValue::ge
```

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`:

```python
from tornado import gen
from tornado import ioloop
from tchannel import TChannel
from tchannel import thrift_request_buidler

from service import KeyValue

KeyValueClient = thrift_request_builder(
    service='keyvalue-server',
    thrift_module=KeyValue,
)

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

    tchannel = TChannel(app_name)
    tchannel.advertise(routers=['127.0.0.1:21300'])

    yield tchannel.thrift(
        KeyValueClient.setValue("foo", "Hello, world!"),
    )

    response = yield tchannel.thrift(
        KeyValueClient.getValue("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.`**`TChannel`**(*name*, *hostport=None*, *process_name=None*, *known_peers=None*, *trace=False*)

Manages connections and requests to other TChannel services.

Usage for a JSON client/server:

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

@tchannel.json.register
def handler(request):
    return {'foo': 'bar'}

response = yield tchannel.json(
    service='some-service',
    endpoint='endpoint',
    headers={'req': 'headers'},
    body={'req': 'body'},
)
```

Variables

- **`thrift`** (ThriftArgScheme) – Make Thrift requests over TChannel and register Thrift handlers.
- **`json`** (JsonArgScheme) – Make JSON requests over TChannel and register JSON handlers.
- **`raw`** (RawArgScheme) – Make requests and register handles that pass raw bytes.

`__init__`(*name*, *hostport=None*, *process_name=None*, *known_peers=None*, *trace=False*)

Note: In general only one `TChannel` instance should be used at a time. Multiple `TChannel` instances are not advisable and could result in undefined behavior.

Parameters

- **`name`** (*string*) – How this application identifies itself. This is the name callers will use to make contact, it is also what your downstream services will see in their metrics.
- **`hostport`** (*string*) – An optional host/port to serve on, e.g., `"127.0.0.1:5555`. If not provided an ephemeral port will be used. When advertising on Hyperbahn you callers do not need to know your port.

call (**args*, ***kwargs*)
Make low-level requests to TChannel services.

Note: Usually you would interact with a higher-level arg scheme like `tchannel.schemes.JsonArgScheme` or `tchannel.schemes.ThriftArgScheme`.

advertise (**args*, ***kwargs*)
Advertise with Hyperbahn.

After a successful advertisement, Hyperbahn will establish long-lived connections with your application. These connections are used to load balance inbound and outbound requests to other applications on the Hyperbahn network.

Re-advertisement happens periodically after calling this method (every minute). Hyperbahn will eject us from the network if it doesn't get a re-advertise from us after 5 minutes.

Parameters

- **routers** (*list*) – A seed list of known Hyperbahn addresses to attempt contact with. Entries should be of the form `"host:port"`.
- **name** (*string*) – The name your application identifies itself as. This is usually unneeded because in the common case it will match the `name` you initialized the `TChannel` instance with. This is the identifier other services will use to make contact with you.
- **timeout** – The timeout (in seconds) for the initial advertise attempt. Defaults to 30 seconds.

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

Raises TimeoutError When unable to make our first advertise request to Hyperbahn. Subsequent requests may fail but will be ignored.

class `tchannel.`**Request** (*body=None*, *headers=None*, *transport=None*, *endpoint=None*)
A TChannel request.

This is sent by callers and received by registered handlers.

Variables

- **body** – The payload of this request. The type of this attribute depends on the scheme being used (e.g., JSON, Thrift, etc.).
- **headers** – A dictionary of application headers. This should be a mapping of strings to strings.
- **transport** – Protocol-level transport headers. These are used for routing over Hyperbahn.

  The most useful piece of information here is probably `request.transport.caller_name`, which is the identity of the application that created this request.

class `tchannel.`**Response** (*body=None*, *headers=None*, *transport=None*, *status=None*)
A TChannel response.

This is sent by handlers and received by callers.

Variables

- **body** – The payload of this response. The type of this attribute depends on the scheme being used (e.g., JSON, Thrift, etc.).
- **headers** – A dictionary of application headers. This should be a mapping of strings to strings.

- **transport** – Protocol-level transport headers. These are used for routing over Hyperbahn.

2.2 Serialization Schemes

2.2.1 Thrift

class tchannel.schemes.**ThriftArgScheme**(*tchannel*)

Handler registration and serialization for Thrift.

To register a Thrift handler:

```
@tchannel.thrift(GeneratedThriftModule)
def method(request):
    print request.body.some_arg
```

When calling a remote service, generated Thrift types need to be wrapped with `thrift_request_builder()` to provide TChannel compatibility:

```
thrift_service = thrift_request_builder(
    service='service-identifier',
    thrift_module=GeneratedThriftModule,
)

response = yield tchannel.thrift(
    thrift_service.method(some_arg='foo'),
)
```

tchannel.**thrift_request_builder**(*service*, *thrift_module*, *hostport=None*, *thrift_class_name=None*)

Provide TChannel compatibility with Thrift-generated modules.

The service this creates is meant to be used with TChannel like so:

```
from tchannel import TChannel, thrift_request_builder
from some_other_service_thrift import some_other_service

tchannel = TChannel('my-service')

some_service = thrift_request_builder(
    service='some-other-service',
    thrift_module=some_other_service
)

resp = tchannel.thrift(
    some_service.fetchPotatoes()
)
```

Parameters

- **service** (*string*) – Name of Thrift service to call. This is used internally for grouping and stats, but also to route requests over Hyperbahn.
- **thrift_module** – The top-level module of the Apache Thrift generated code for the service that will be called.
- **hostport** (*string*) – When calling the Thrift service directly, and not over Hyperbahn, this 'host:port' value should be provided.

- **`thrift_class_name`** (*string*) – When the Apache Thrift generated Iface class name does not match thrift_module, then this should be provided.

2.2.2 JSON

class `tchannel.schemes.`**`JsonArgScheme`**(*tchannel*)

Semantic params and serialization for json.

`__call__`(**args*, ***kwargs*)

Make JSON TChannel Request.

Parameters

- **`service`** (*string*) – Name of the service to call.
- **`endpoint`** (*string*) – Endpoint to call on service.
- **`body`** (*string*) – A raw body to provide to the endpoint.
- **`headers`** (*string*) – A raw headers block to provide to the endpoint.
- **`timeout`** (*int*) – How long to wait (in ms) before raising a `TimeoutError` - this defaults to `tchannel.glossary.DEFAULT_TIMEOUT`.
- **`retry_on`** (*string*) – What events to retry on - valid values can be found in `tchannel.retry`.
- **`retry_limit`** (*string*) – How many times to retry before
- **`hostport`** (*string*) – A 'host:port' value to use when making a request directly to a TChannel service, bypassing Hyperbahn.

Return type *Response*

2.2.3 Raw

class `tchannel.schemes.`**`RawArgScheme`**(*tchannel*)

Semantic params and serialization for raw.

`__call__`(*service*, *endpoint*, *body=None*, *headers=None*, *timeout=None*, *retry_on=None*, *retry_limit=None*, *hostport=None*, *shard_key=None*, *trace=None*)

Make a raw TChannel request.

The request's headers and body are treated as raw bytes and not serialized/deserialized.

The request's headers and body are treated as raw bytes and not serialized/deserialized.

Parameters

- **`service`** (*string*) – Name of the service to call.
- **`endpoint`** (*string*) – Endpoint to call on service.
- **`body`** (*string*) – A raw body to provide to the endpoint.
- **`headers`** (*string*) – A raw headers block to provide to the endpoint.
- **`timeout`** (*int*) – How long to wait (in ms) before raising a `TimeoutError` - this defaults to `tchannel.glossary.DEFAULT_TIMEOUT`.
- **`retry_on`** (*string*) – What events to retry on - valid values can be found in `tchannel.retry`.

- **`retry_limit`** (*string*) – How many times to retry before
- **`hostport`** (*string*) – A 'host:port' value to use when making a request directly to a TChannel service, bypassing Hyperbahn.

Return type *Response*

2.3 Exception Handling

2.3.1 Errors

`tchannel.errors.`**`TIMEOUT`** **= 1**
The request timed out.

`tchannel.errors.`**`CANCELED`** **= 2**
The request was canceled.

`tchannel.errors.`**`BUSY`** **= 3**
The server was busy.

`tchannel.errors.`**`BAD_REQUEST`** **= 6**
The request was bad.

`tchannel.errors.`**`NETWORK_ERROR`** **= 7**
There was a network error when sending the request.

`tchannel.errors.`**`UNHEALTHY`** **= 8**
The server handling the request is unhealthy.

`tchannel.errors.`**`FATAL`** **= 255**
There was a fatal protocol-level error.

exception `tchannel.errors.`**`TChannelError`**(*description=None*, *id=None*, *tracing=None*)
Bases: `exceptions.Exception`

A TChannel-generated exception.

Variables **`code`** – The error code for this error. See the Specification for a description of these codes.

classmethod **`from_code`**(*code*, ***kw*)
Construct a `TChannelError` instance from an error code.

This will return the appropriate class type for the given code.

exception `tchannel.errors.`**`RetryableError`**(*description=None*, *id=None*, *tracing=None*)
Bases: *`tchannel.errors.TChannelError`*

An error where the original request is always safe to retry.

It is always safe to retry a request with this category of errors. The original request was never handled.

exception `tchannel.errors.`**`MaybeRetryableError`**(*description=None*, *id=None*, *tracing=None*)
Bases: *`tchannel.errors.TChannelError`*

An error where the original request may be safe to retry.

The original request may have reached the intended service. Hence, the request should only be retried if it is known to be idempotent.

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

Bases: *`tchannel.errors.TChannelError`*

An error where the original request should not be re-sent.

Something was fundamentally wrong with the request and it should not be retried.

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

Bases: `tchannel.errors.FatalProtocolError`

Raised when there is an error while reading input.

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

Bases: `tchannel.errors.FatalProtocolError`

Represent invalid checksum type in the message

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

Bases: *`tchannel.errors.RetryableError`*

Represents a failure to find any peers for a request.

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

Bases: `tchannel.errors.FatalProtocolError`

Raised when attempting to listen multiple times.

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

Bases: `tchannel.errors.BadRequestError`

Raised when a one-way Thrift procedure is called.

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

Bases: `tchannel.errors.BadRequestError`

Raised when a non-void Thrift response contains no value.

2.3.2 Retry Behavior

These values can be passed as the `retry_on` behavior to *`tchannel.TChannel.call()`*.

`tchannel.retry.`**`CONNECTION_ERROR`** **= u'c'**

Retry the request on failures to connect to a remote host. This is the default retry behavior.

`tchannel.retry.`**`NEVER`** **= u'n'**

Never retry the request.

`tchannel.retry.`**`TIMEOUT`** **= u't'**

Retry the request on timeouts waiting for a response.

`tchannel.retry.`**`CONNECTION_ERROR_AND_TIMEOUT`** **= u'ct'**

Retry the request on failures to connect and timeouts after connecting.

`tchannel.retry.`**`DEFAULT_RETRY_LIMIT`** **= 4**

The default number of times to retry a request. This is in addition to the original request.

2.4 Synchronous Client

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

Make synchronous TChannel requests.

This client does not support incoming requests – it is a uni-directional client only.

The client is implemented on top of the Tornado-based implementation and offloads IO to a thread running an `IOLoop` next to your process.

Usage mirrors the `TChannel` class.

```
tchannel = TChannel(name='my-synchronous-service')

# Advertise with Hyperbahn.
# This returns a future. You may want to block on its result,
# particularly if you want you app to die on unsuccessful
# advertisement.
tchannel.advertise(routers)

# thrift_service is the result of a call to ``thrift_request_builder``
future = tchannel.thrift(
    thrift_service.getItem('foo'),
    timeout=1000,
)

result = future.result()
```

`advertise`(**args*, ***kwargs*)

Advertise with Hyperbahn.

After a successful advertisement, Hyperbahn will establish long-lived connections with your application. These connections are used to load balance inbound and outbound requests to other applications on the Hyperbahn network.

Re-advertisement happens periodically after calling this method (every minute). Hyperbahn will eject us from the network if it doesn't get a re-advertise from us after 5 minutes.

Parameters

- **`routers`** (*list*) – A seed list of known Hyperbahn addresses to attempt contact with. Entries should be of the form `"host:port"`.
- **`name`** (*string*) – The name your application identifies itself as. This is usually unneeded because in the common case it will match the `name` you initialized the `TChannel` instance with. This is the identifier other services will use to make contact with you.
- **`timeout`** – The timeout (in seconds) for the initial advertise attempt. Defaults to 30 seconds.

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

Raises TimeoutError When unable to make our first advertise request to Hyperbahn. Subsequent requests may fail but will be ignored.

`call`(**args*, ***kwargs*)

Make low-level requests to TChannel services.

Note: Usually you would interact with a higher-level arg scheme like *`tchannel.schemes.JsonArgScheme`* or *`tchannel.schemes.ThriftArgScheme`*.

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.

```python
from tchannel.testing import vcr

@pytest.mark.gen_test
@vcr.use_cassette('tests/data/bar.yaml')
def test_bar():
    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.

CHAPTER 3

Changelog

3.1 Changes by Version

3.1.1 0.16.6 (2015-09-14)

- Fixed a bug where Zipkin traces were not being propagated correctly in services using the `tchannel.TChannel` API.

3.1.2 0.16.5 (2015-09-09)

- Actually fix status code being unset in responses when using the Thrift scheme.
- Fix request TTLs not being propagated over the wire.

3.1.3 0.16.4 (2015-09-09)

- Fix bug where status code was not being set correctly on call responses for application errors when using the Thrift scheme.

3.1.4 0.16.3 (2015-09-09)

- Make `TChannel.listen` thread-safe and idempotent.

3.1.5 0.16.2 (2015-09-04)

- Fix *retry_limit* in *TChannel.call* not allowing 0 retries.

3.1.6 0.16.1 (2015-08-27)

- Fixed a bug where the 'not found' handler would incorrectly return serialization mismatch errors..
- Fixed a bug which prevented VCR support from working with the sync client.
- Fixed a bug in VCR that prevented it from recording requests made by the sync client, and requests made with `hostport=None`.
- Made `client_for` compatible with `tchannel.TChannel`.

- Brought back `tchannel.sync.client_for` for backwards compatibility.

3.1.7 0.16.0 (2015-08-25)

- Introduced new server API through methods `tchannel.TChannel.thrift.register`, `tchannel.TChannel.json.register`, and `tchannel.TChannel.raw.register` - when these methods are used, endpoints are passed a `tchannel.Request` object, and are expected to return a `tchannel.Response` object or just a response body. The deprecated `tchannel.tornado.TChannel.register` continues to function how it did before. Note the breaking change to the top-level TChannel on the next line.
- Fixed a crash that would occur when forking with an unitialized `TChannel` instance.
- Add `hooks` property in the `tchannel.TChannel` class.
- **BREAKING** - `tchannel.TChannel.register` no longer has the same functionality as `tchannel.tornado.TChannel.register`, instead it exposes the new server API. See the upgrade guide for details.
- **BREAKING** - remove `retry_delay` option in the `tchannel.tornado.send` method.
- **BREAKING** - error types have been reworked significantly. In particular, the all-encompassing `ProtocolError` has been replaced with more granualar/actionable exceptions. See the upgrade guide for more info.
- **BREAKING** - Remove third `proxy` argument from the server handler interface.
- **BREAKING** - `ZipkinTraceHook` is not longer registered by default.
- **BREAKING** - `tchannel.sync.client.TChannelSyncClient` replaced with `tchannel.sync.TChannel`.

3.1.8 0.15.2 (2015-08-07)

- Raise informative and obvious `ValueError` when anything but a map[string]string is passed as headers to the `TChannel.thrift` method.
- First param, request, in `tchannel.thrift` method is required.

3.1.9 0.15.1 (2015-08-07)

- Raise `tchannel.errors.ValueExpectedError` when calling a non-void Thrift procedure that returns no value.

3.1.10 0.15.0 (2015-08-06)

- Introduced new top level `tchannel.TChannel` object, with new request methods `call`, `raw`, `json`, and `thrift`. This will eventually replace the akward `request` / `send` calling pattern.
- Introduced `tchannel.thrift_request_builder` function for creating a request builder to be used with the `tchannel.TChannel.thrift` function.
- Introduced new simplified examples under the `examples/simple` directory, moved the Guide's examples to `examples/guide`, and deleted the remaining examples.
- Added ThriftTest.thrift and generated Thrift code to `tchannel.testing.data` for use with examples and playing around with TChannel.

- Fix JSON arg2 (headers) being returned a string instead of a dict.

3.1.11 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`.

3.1.12 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.

3.1.13 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`.

3.1.14 0.11.2 (2015-07-20)

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

3.1.15 0.11.1 (2015-07-17)

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

3.1.16 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`.

3.1.17 0.10.3 (2015-07-13)

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

3.1.18 0.10.2 (2015-07-13)

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

3.1.19 0.10.1 (2015-07-10)

- Previous release was broken with older versions of pip.

3.1.20 0.10.0 (2015-07-10)

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

3.1.21 0.9.1 (2015-07-09)

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

3.1.22 0.9.0 (2015-07-07)

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

3.1.23 0.8.5 (2015-06-30)

- Add port parameter for `TChannel.listen`.

3.1.24 0.8.4 (2015-06-17)

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

3.1.25 0.8.3 (2015-06-15)

- Add `as` attribute to the response header.

3.1.26 0.8.2 (2015-06-11)

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

3.1.27 0.8.1 (2015-06-10)

- Initial release.

3.2 Upgrade Guide

Migrating to a version of TChannel with breaking changes? This guide documents what broke and how to safely migrate to newer versions.

3.2.1 From 0.15 to 0.16

- `tchannel.TChannel.register` no longer mimicks `tchannel.tornado.TChannel.register`, instead it exposes the new server API like so:

  Before:

```
from tchannel.tornado import TChannel

tchannel = TChannel('my-service-name')

@tchannel.register('endpoint', 'json')
def endpoint(request, response, proxy):
    response.write({'resp': 'body'})
```

  After:

```
from tchannel import TChannel

tchannel = TChannel('my-service-name')

@tchannel.json.register
def endpoint(request):
    return {'resp': 'body'}

    # Or, if you need to return headers with your response:
    from tchannel import Response
    return Response({'resp': 'body'}, {'header': 'foo'})
```

- `TChannelSyncClient` has been replaced with `tchannel.sync.TChannel`. This new synchronous client has been significantly re-worked to more closely match the asynchronous `TChannel` API. `tchannel.sync.thrift.client_for` has been removed and `tchannel.thrift_request_builder` should be used instead (`tchannel.thrift.client_for` still exists for backwards compatibility but is not recommended). This new API allows specifying headers, timeouts, and retry behavior with Thrift requests.

  Before:

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

from generated.thrift.code import MyThriftService

tchannel_thrift_client = client_for('foo', MyThriftService)

tchannel = TChannelSyncClient(name='bar')

future = tchannel_thrift_client.someMethod(...)

result = future.result()
```

  After:

```
from tchannel import thrift_request_builder
from tchannel.sync import TChannel
from tchannel.retry import CONNECTION_ERROR_AND_TIMEOUT

from generated.thrift.code import MyThriftService

tchannel_thrift_client = thrift_request_builder(
    service='foo',
    thrift_module=MyThriftService,
)

tchannel = TChannel(name='bar')

future = tchannel.thrift(
    tchannel_thrift_client.someMethod(...)
    headers={'foo': 'bar'},
    retry_on=CONNECTION_ERROR_AND_TIMEOUT,
    timeout=1000,
)

result = future.result()
```

- `from tchannel.tornado import TChannel` is deprecated.
- Removed `retry_delay` option from `tchannel.tornado.peer.PeerClientOperation.send` method.

  Before: `tchannel.tornado.TChannel.request.send(retry_delay=300)`

  After: no more `retry_delay` in `tchannel.tornado.TChannel.request.send()`
- If you were catching `ProtocolError` you will need to catch a more specific type, such as `TimeoutError`, `BadRequestError`, `NetworkError`, `UnhealthyError`, or `UnexpectedError`.
- If you were catching `AdvertiseError`, it has been replaced by `TimeoutError`.
- If you were catching `BadRequest`, it may have been masking checksum errors and fatal streaming errors. These are now raised as `FatalProtocolError`, but in practice should not need to be handled when interacting with a well-behaved TChannel implementation.
- `TChannelApplicationError` was unused and removed.
- Three error types have been introduced to simplify retry handling: - `NotRetryableError` (for requests should never be retried), - `RetryableError` (for requests that are always safe to retry), and - `MaybeRetryableError` (for requests that are safe to retry on idempotent

    endpoints).

3.2.2 From 0.14 to 0.15

- No breaking changes.

3.2.3 From 0.13 to 0.14

- No breaking changes.

3.2.4 From 0.12 to 0.13

- No breaking changes.

3.2.5 From 0.11 to 0.12

- Removed `print_arg`. Use `request.get_body()` instead.

3.2.6 From 0.10 to 0.11

- Renamed `tchannel.tornado.TChannel.advertise` argument `router` to `routers`. Since this is a required arg and the first positional arg, only clients who are using as kwarg will break.

  Before: `tchannel.advertise(router=['localhost:21300'])`

  After: `tchannel.advertise(routers=['localhost:21300'])`

Index