grpclib

Vladimir Magamedov

Oct 17, 2022

CONTENTS

This project is based on hyper-h2 and **requires Python >= 3.7**.

CHAPTER

ONE

EXAMPLE

See examples directory in the project's repository for all available examples.

1.1 Client

```
import asyncio

from grpclib.client import Channel

# generated by protoc
from .helloworld_pb2 import HelloRequest, HelloReply
from .helloworld_grpc import GreeterStub


async def main():
    async with Channel('127.0.0.1', 50051) as channel:
        greeter = GreeterStub(channel)

        reply = await greeter.SayHello(HelloRequest(name='Dr. Strange'))
        print(reply.message)


if __name__ == '__main__':
    asyncio.run(main())
```

1.2 Server

```
import asyncio

from grpclib.utils import graceful_exit
from grpclib.server import Server

# generated by protoc
from .helloworld_pb2 import HelloReply
from .helloworld_grpc import GreeterBase
```

(continues on next page)

(continued from previous page)

```
class Greeter(GreeterBase):

    async def SayHello(self, stream):
        request = await stream.recv_message()
        message = f'Hello, {request.name}!'
        await stream.send_message(HelloReply(message=message))


async def main(*, host='127.0.0.1', port=50051):
    server = Server([Greeter()])
    # Note: graceful_exit isn't supported in Windows
    with graceful_exit([server]):
        await server.start(host, port)
        print(f'Serving on {host}:{port}')
        await server.wait_closed()


if __name__ == '__main__':
    asyncio.run(main())
```

CHAPTER

TWO

INSTALLATION

```
$ pip3 install "grpclib[protobuf]"
```

Bug fixes and new features are frequently published via release candidates:

```
$ pip3 install --upgrade --pre "grpclib[protobuf]"
```

For the code generation you will also need a `protoc` compiler, which can be installed with `protobuf` system package:

```
$ brew install protobuf  # example for macOS users
$ protoc --version
libprotoc ...
```

Or you can use `protoc` compiler from the `grpcio-tools` Python package:

```
$ pip3 install grpcio-tools
$ python3 -m grpc_tools.protoc --version
libprotoc ...
```

Note: `grpcio` and `grpcio-tools` packages are **not required in runtime**, `grpcio-tools` package will be used only during code generation.

CHAPTER

THREE

PROTOC PLUGIN

In order to use this library you will have to generate special stub files using plugin provided, which can be used like this:

```
$ python3 -m grpc_tools.protoc -I. --python_out=. --grpclib_python_out=. helloworld/
↪helloworld.proto
                                                    ^----- note -----^
```

This command will generate `helloworld_pb2.py` and `helloworld_grpc.py` files.

Plugin which implements `--grpclib_python_out` option should be available for the `protoc` compiler as the `protoc-gen-grpclib_python` executable which should be installed by `pip` into your `$PATH` during installation of the `grpclib` library.

Changed in v0.3.2: `--python_grpc_out` option was renamed into `--grpclib_python_out`.

CHAPTER

FOUR

CONTRIBUTING

- Please submit an issue before working on a Pull Request
- Do not merge/squash/rebase your development branch while you work on a Pull Request, use rebase if this is really necessary
- You may use Tox in order to test and lint your changes, but it is Ok to rely on CI for this matter

4.1 Changelog

4.1.1 0.4.3

- **BREAKING:** Regenerated internal `*_pb2.py` files, they now require `protobuf>=3.15.0`
- Fixed mistake of subclassing `AbstractServer` in `grpclib.server.Server`, this fixes mypy errors
- Fixed `ChannelFor` utility to properly cleanup resources without cryptic tracebacks (see #156)
- Added Python 3.10 support, dropped Python 3.6 support
- Fixed TLS support in Python 3.10; pull request courtesy Scott Phillips @fundthmcalculus
- Fixed setup.cfg to include generated .pyi files

4.1.2 0.4.2

- **BREAKING:** Regenerated internal `*_pb2.py` files, they now require `protobuf>=3.12.0`
- Extended `SendTrailingMetadata` and `RecvTrailingMetadata` events with a status-related properties
- Fixed deprecation warning related to `asyncio.wait()` and Python 3.9
- Added support for the `--experimental_allow_proto3_optional` protoc flag

4.1.3 0.4.1

- Fixed `h2==4.0.0` compatibility, fixed project dependencies

4.1.4 0.4.0

- Fixed `Config._http2_max_pings_without_data` value validation, it may be equal to 0 to send `PING` frames indefinitely
- Added context-manager protocol to the `Channel` class
- **BREAKING:** Fixed metadata validation, this may cause an exceptions when you try to send invalid metadata values
- Added certifi support, documented secure channels
- Added `http2_connection_window_size` and `http2_stream_window_size` config values, using 4 MiB as a default for both values instead of 64 KiB (HTTP/2 default)

4.1.5 0.3.2

- Using `application/grpc` content type on the client-side to be compatible with faulty server implementations (e.g. googleapis.com)
- Renamed `--python_grpc_out=` protoc option into `--grpclib_python_out=` to avoid misunderstanding and to follow grpc project naming
- Added `(client|server):Stream.peer` property and corresponding property in the `RecvRequest` event
- Added `server:Stream.user_agent` property and corresponding property in the `RecvRequest` event
- Fixed `time.monotonic()` usage in the `ServiceCheck` class for the case when monotonic time starts from zero
- Fixed `release_stream()` function to not send data over a connection if connection is already closed
- Implement connection checks using PING frame (experimental); pull request courtesy Evhenii Popovych @a00920
- Fixed code generation plugin when a path to proto file contains hyphens and/or `.protodevel` file extension; pull request courtesy @linw1995
- Deprecated `loop` argument in a public APIs
- Exposed new import locations for the `Status` and `GRPCError` classes:

```
from grpclib import Status, GRPCError
```

- Added `grpclib.config.Configuration` class to configure grpclib internals (experimental)
- Disabled unnecessary and expensive headers validation and normalization

4.1.6 0.3.1

- Fixed code generation plugin to support nested message types in a service definitions
- Restored `v1alpha` reflection protocol support
- Implemented "grpc-status-details-bin" metadata support

4.1.7 0.3.0

- Lowered log level for successfully handled errors on the server-side
- Turned assert statement into `TypeError` in the `ProtoCodec.encode` method
- Raising proper `GRPCError` in `client.Stream.__aexit__` after receiving `RST_STREAM` frame
- Logging protocol errors, caused by the other side
- Removed `v1alpha` reflection protocol, `v1` remains
- Added example of using `ProcessPoolExecutor` for CPU-intensive tasks
- Covered library and examples with type annotations, many thanks and credit to Callum Ryan @c-ryan747 for his work on #64
- Fixed implicit trailers-only response for streaming calls
- Added `end` argument to the `client.Stream.send_request` method
- **BREAKING:** Removed deprecated `end` argument from the `server.Stream.send_message` method
- Fixed server to send content-type header in a trailers-only responses
- Implemented support for the trailers-only empty response on the client-side
- Made `loop` argument optional in a user-facing apis
- Added more checks to verify that streams are used accordingly to the gRPC protocol spec
- **BREAKING:** Undocumented `Channel.request` method was changed in a backward-incompatible way
- Dropped Python 3.5 support for async generators and better typing support
- **BREAKING:** Removed undocumented `grpclib.metadata.Metadata` class
- Implemented ability to listen for "events" from grpclib, see *Events* for more information

4.1.8 0.2.5

- Fixed `protocol.Stream.send_data` method to properly wait for a positive window size

4.1.9 0.2.4

- Fixed and refactored protocol.Buffer class to properly acknowledge received data, which is critical for flow control mechanism. Also added logic to acknowledge all unread by user data before and after stream release.

4.1.10 0.2.3

- Removed circular references and added tests to detect them
- Generate `*_grpc.py` stub files even if service definitions don't exist in the .proto files
- Fixed bug in the Channel.request method, deadline argument was ignored
- Implemented `graceful_exit` context-manager

4.1.11 0.2.2

- Logging StreamTerminatedError on the server-side if client resets stream
- Improved health checks support
- Stream methods now can be called concurrently
- Fixed flow-control window change detection for the case when the other party relies on connection-level window with unlimited stream-level windows
- Fixed PING frame support on the server-side

4.1.12 0.2.1

- Added `Channel.__del__` method to close unclosed connections and warn about them
- Changed user-agent header to reflect `grpclib` and Python versions
- Added workaround for `h2`, when `h2` raises `StreamIDTooLowError` instead of `StreamClosedError`
- Fixed race condition in the `Channel`, which leads to creation of more than one connection
- Fixed Python 3.5.1 compatibility

4.1.13 0.2.0

- Fixed flow control functionality
- Generate `*_grpc.py` stub files only if service definitions exists in the .proto files
- Fixed possibility of the infinite loop when we reach max outbound streams limit and wait for a closed stream during `grpclib.protocol.Stream.send_request()` method call
- Added support for secure channels through SSL/TLS; pull request courtesy Michael P. Nitowski @mnito
- Implemented Health service with additional functionality to help write health checks
- Implemented `ChannelFor` helper for writing functional tests
- Added support for UNIX sockets; pull request courtesy Andy Kipp @kippandrew
- Implemented server reflection protocol
- **BREAKING:** Fixed metadata encoding. Previously grpclib were using utf-8 to encode metadata, and now grpclib encodes metadata according to the gRPC wire protocol specification: ascii for regular values and base64 for binary values
- **BREAKING:** Fixed "grpc-message" header encoding: unicode string -> utf-8 -> percent-encoding (RFC 3986, ascii subset). Previously solely utf-8 were used, which now will fail to decode, if you send non-ascii characters
- Implemented sending custom metadata from the server-side

4.1.14 0.1.1

- Dropped protobuf requirement, now it's optional
- New feature to specify custom message serialization/deserialization codec
- Fixed critical issue on the client-side with hanging coroutines in case of connection lost or stream reset
- Replaced `async-timeout` dependency with custom utilities, refactored deadlines implementation
- Improved connection lost handling; pull request courtesy Michael Elsdörfer @miracle2k
- Improved error responses and errors handling
- Deprecated `end` keyword-only argument in the *grpclib.server.Stream.send_message()* method on the server-side

4.1.15 0.1.0

- Improved example to show all RPC method types; pull request courtesy @claws
- [rc2] Fixed issues with sending large messages
- [rc1] Initial release

4.2 Overview

gRPC protocol is exclusively based on HTTP/2 (aka h2) protocol. Main concepts:

- each request in h2 connection is a bidirectional stream of frames;
- streams give ability to do multiplexing - make requests in parallel using single TCP connection;
- h2 has special flow control mechanism, which can help avoid network congestion and fix problems with slow clients, slow servers and slow network;
- flow control only affects `DATA` frames, any other frame can be sent without limitations;
- streams can be cancelled individually, all other streams in h2 connection will continue work and there is no need to drop connection and reconnect;
- h2 is a binary protocol and allows headers compression using HPACK format, so it is a very strict and efficient protocol.

h2 protocol is highly configurable, for example:

- flow control mechanism can use dynamically configurable initial window size, to better match different use cases and conditions;
- you can set maximum frame size to control how much data you will receive in each frame;
- you can limit number of concurrent streams for h2 connection.

gRPC protocol adds to h2 protocol messages encoding format and a notion about metadata. gRPC metadata == additional h2 headers. So gRPC has the same level of extensibility as HTTP has.

Messages are sent using one or several `DATA` frames, depending on maximum frame size setting and message size. Messages are encoded using simple format: prefix + data. Prefix contains length of the data and compression flag. You can learn gRPC wire protocol in more details here: gRPC format.

gRPC has 4 method types: unary-unary, unary-stream (e.g. download), stream-unary (e.g. upload), stream-stream. They are all the same, the only difference is how many messages are sent in each direction: exactly one (unary) or any number of messages (stream).

4.2.1 Cancellation

As it was said above, h2 allows you to cancel any stream without affecting other streams, which are living in the same connection. And h2 protocol has special frame to do this: RST_STREAM. Both client and server can cancel streams. This feature automatically gives you ability to proactively cancel gRPC method calls in the same way. In `grpclib` you can cancel method calls immediately, for example:

- client sends request to the server
- server spawns task to handle this request
- client wants to cancel this request and sends `RST_STREAM` frame
- server receives `RST_STREAM` frame and cancels task immediately

Most other protocols doesn't have this feature, so they have to terminate whole TCP connection and perform reconnect for the next call. It is also not obvious how to immediately detect terminated connections on the other side, and this means that server most likely will continue result computations, when this result is not needed anymore.

4.2.2 Deadlines

Deadlines are basically timeouts, which are propagated from service to service, to meet initial timeout constrains. This is a simple and powerful idea.

Example:

- service X receives request with `grpc-timeout: 100m` in metadata (100m means 100 milliseconds)
  - service X immediately converts timeout into deadline:

```
deadline = time.monotonic() + grpc_timeout
```

  - service X spent 20ms doing some work
  - now service X wants to make outgoing request to service Y, so it computes how much time remains to perform this request:

```
new_timeout = max(deadline - time.monotonic(), 0)  # == 80ms
```

  - service X performs request to service Y with metadata `grpc-timeout: 80m`
    - service Y uses the same logic to convert timeout -> deadline -> timeout.

With this feature it is possible to cancel whole call chain simultaneously, even in case of network failures (broken connections).

4.2.3 grpclib

`grpclib` tries to give you full control over these bidirectional h2 streams.

Note: *[auto]* mark below means that it is not necessary to explicitly call these methods in your code, they will be called automatically behind the scenes. They are exists to have more control.

4.3 Client

A single *Channel* represents a single connection to the server. Because gRPC is based on HTTP/2, there is no need to create multiple connections to the server, many concurrent RPC calls can be performed through a single multiplexed connection. See *Overview* for more details.

```
async with Channel(host, port) as channel:
    pass
```

A single server can implement several services, so you can reuse one channel for all corresponding service stubs:

```
foo_svc = FooServiceStub(channel)
bar_svc = BarServiceStub(channel)
baz_svc = BazServiceStub(channel)
```

There are two ways to call RPC methods:

- simple, suitable for unary-unary calls:

  ```
  reply = await stub.Method(Request())
  ```

- advanced, suitable for streaming calls:

  ```
  async with stub.BiDiMethod.open() as stream:
      await stream.send_request()  # needed to initiate a call
      while True:
          task = await task_queue.get()
          if task is None:
              await stream.end()
              break
          else:
              await stream.send_message(task)
              result = await stream.recv_message()
              await result_queue.add(task)
  ```

See reference docs for all method types and for the *Stream* methods and attributes.

4.3.1 Secure Channels

Here is how to establish a secure connection to a public gRPC server:

```
channel = Channel(host, port, ssl=True)
                              ^^^^^^^^
```

In this case `grpclib` uses system CA certificates. But `grpclib` has also a built-in support for a certifi package which contains actual Mozilla's collection of CA certificates. All you need is to install it and keep it updated – this is a more favorable way than relying on system CA certificates:

```
$ pip3 install certifi
```

`grpclib` also allows you to use a custom SSL configuration by providing a `SSLContext` object. We have a simple mTLS auth example in our code repository to illustrate how this works.

4.3.2 Reference

class grpclib.client.**Stream**(*channel: Channel*, *method_name: str*, *metadata: _Metadata*, *cardinality: Cardinality*, *send_type: Type[_SendType]*, *recv_type: Type[_RecvType]*, ***, *codec: CodecBase*, *status_details_codec: Optional[StatusDetailsCodecBase]*, *dispatch: _DispatchChannelEvents*, *deadline: Optional[Deadline] = None*)

Represents gRPC method call - HTTP/2 request/stream, and everything you need to communicate with server in order to get response.

In order to work directly with stream, you should `ServiceMethod.open()` request like this:

```
request = cafe_pb2.LatteOrder(
    size=cafe_pb2.SMALL,
    temperature=70,
    sugar=3,
)
async with client.MakeLatte.open() as stream:
    await stream.send_message(request, end=True)
    reply: empty_pb2.Empty = await stream.recv_message()
```

initial_metadata: Optional[_Metadata] = None

This property contains initial metadata, received with headers from the server. It equals to `None` initially, and to a multi-dict object after *recv_initial_metadata()* coroutine succeeds.

trailing_metadata: Optional[_Metadata] = None

This property contains trailing metadata, received with trailers from the server. It equals to `None` initially, and to a multi-dict object after *recv_trailing_metadata()* coroutine succeeds.

peer: Optional[*Peer*] = None

Connection's peer info of type *Peer*

async send_request(***, *end: bool = False*) → None

Coroutine to send request headers with metadata to the server.

New HTTP/2 stream will be created during this coroutine call.

Note: This coroutine will be called implicitly during first *send_message()* coroutine call, if not called before explicitly.

Parameters
end – end outgoing stream if there are no messages to send in a streaming request

async send_message(*message: _SendType*, ***, *end: bool = False*) → None

Coroutine to send message to the server.

If client sends UNARY request, then you should call this coroutine only once. If client sends STREAM request, then you can call this coroutine as many times as you need.

> **Warning:** It is important to finally end stream from the client-side when you finished sending messages.

You can do this in two ways:

- specify `end=True` argument while sending last message - and last DATA frame will include END_STREAM flag;
- call *end()* coroutine after sending last message - and extra HEADERS frame with END_STREAM flag will be sent.

First approach is preferred, because it doesn't require sending additional HTTP/2 frame.

async end() → None

Coroutine to end stream from the client-side.

It should be used to finally end stream from the client-side when we're finished sending messages to the server and stream wasn't closed with last DATA frame. See *send_message()* for more details.

HTTP/2 stream will have half-closed (local) state after this coroutine call.

async recv_initial_metadata() → None

Coroutine to wait for headers with initial metadata from the server.

Note: This coroutine will be called implicitly during first *recv_message()* coroutine call, if not called before explicitly.

May raise *GRPCError* if server returned non-*Status.OK* in trailers-only response.

When this coroutine finishes, you can access received initial metadata by using *initial_metadata* attribute.

async recv_message() → Optional[_RecvType]

Coroutine to receive incoming message from the server.

If server sends UNARY response, then you can call this coroutine only once. If server sends STREAM response, then you should call this coroutine several times, until it returns None when the server has ended the stream. To simplify you code in this case, *Stream* implements async iterations protocol, so you can use it like this:

```
async for message in stream:
    do_smth_with(message)
```

or even like this:

```
messages = [msg async for msg in stream]
```

HTTP/2 has flow control mechanism, so client will acknowledge received DATA frames as a message only after user consumes this coroutine.

Returns
message

async recv_trailing_metadata() → None

Coroutine to wait for trailers with trailing metadata from the server.

Note: This coroutine will be called implicitly at exit from this call (context manager's exit), if not called before explicitly.

May raise *GRPCError* if server returned non-*Status.OK* in trailers.

When this coroutine finishes, you can access received trailing metadata by using *trailing_metadata* attribute.

async cancel() → None

Coroutine to cancel this request/stream.

Client will send RST_STREAM frame to the server, so it will be explicitly informed that there is nothing to expect from the client regarding this request/stream.

class grpclib.client.**Channel**(*host: Optional[str] = None*, *port: Optional[int] = None*, *, *loop: Optional[AbstractEventLoop] = None*, *path: Optional[str] = None*, *codec: Optional[CodecBase] = None*, *status_details_codec: Optional[StatusDetailsCodecBase] = None*, *ssl: Union[None, bool, SSLContext] = None*, *config: Optional[*Configuration*] = None*)

Represents a connection to the server, which can be used with generated stub classes to perform gRPC calls.

```
channel = Channel()
client = cafe_grpc.CoffeeMachineStub(channel)

...

request = cafe_pb2.LatteOrder(
    size=cafe_pb2.SMALL,
    temperature=70,
    sugar=3,
)
reply: empty_pb2.Empty = await client.MakeLatte(request)

...

channel.close()
```

Initialize connection to the server

Parameters

- **host** – server host name.
- **port** – server port number.
- **loop** – (deprecated) asyncio-compatible event loop
- **path** – server socket path. If specified, host and port should be omitted (must be None).

- **codec** – instance of a codec to encode and decode messages, if omitted `ProtoCodec` is used by default
- **status_details_codec** – instance of a status details codec to decode error details in a trailing metadata, if omitted `ProtoStatusDetailsCodec` is used by default
- **ssl** – `True` or `SSLContext` object; if `True`, default SSL context is used.

close() → None

Closes connection to the server.

class grpclib.client.**UnaryUnaryMethod**(*channel: Channel*, *name: str*, *request_type: Type[_SendType]*, *reply_type: Type[_RecvType]*)

Represents UNARY-UNARY gRPC method type.

async **__call__**(*message: _SendType*, ***, *timeout: Optional[float] = None*, *metadata: Optional[Union[Mapping[str, Union[str, bytes]], Collection[Tuple[str, Union[str, bytes]]]]] = None*) → _RecvType

Coroutine to perform defined call.

Parameters

- **message** – message
- **timeout** (*float*) – request timeout (seconds)
- **metadata** – custom request metadata, dict or list of pairs

Returns
message

open(***, *timeout: Optional[float] = None*, *metadata: Optional[Union[Mapping[str, Union[str, bytes]], Collection[Tuple[str, Union[str, bytes]]]]] = None*) → *Stream*[_SendType, _RecvType]

Creates and returns *Stream* object to perform request to the server.

Nothing will happen to the current underlying HTTP/2 connection during this method call. It just initializes *Stream* object for you. Actual request will be sent only during *Stream.send_request()* or *Stream.send_message()* coroutine call.

Parameters

- **timeout** (*float*) – request timeout (seconds)
- **metadata** – custom request metadata, dict or list of pairs

Returns
Stream object

class grpclib.client.**UnaryStreamMethod**(*channel: Channel*, *name: str*, *request_type: Type[_SendType]*, *reply_type: Type[_RecvType]*)

Represents UNARY-STREAM gRPC method type.

async **__call__**(*message: _SendType*, ***, *timeout: Optional[float] = None*, *metadata: Optional[Union[Mapping[str, Union[str, bytes]], Collection[Tuple[str, Union[str, bytes]]]]] = None*) → List[_RecvType]

Coroutine to perform defined call.

Parameters

- **message** – message
- **timeout** (*float*) – request timeout (seconds)

- **metadata** – custom request metadata, dict or list of pairs

Returns
sequence of messages

open(*, *timeout: Optional[float] = None*, *metadata: Optional[Union[Mapping[str, Union[str, bytes]], Collection[Tuple[str, Union[str, bytes]]]]] = None*) → *Stream*[_SendType, _RecvType]

Creates and returns *Stream* object to perform request to the server.

Nothing will happen to the current underlying HTTP/2 connection during this method call. It just initializes *Stream* object for you. Actual request will be sent only during *Stream.send_request()* or *Stream.send_message()* coroutine call.

Parameters

- **timeout** (*float*) – request timeout (seconds)
- **metadata** – custom request metadata, dict or list of pairs

Returns
Stream object

class grpclib.client.**StreamUnaryMethod**(*channel: Channel*, *name: str*, *request_type: Type[_SendType]*, *reply_type: Type[_RecvType]*)

Represents STREAM-UNARY gRPC method type.

async __call__(*messages: Sequence[_SendType]*, *, *timeout: Optional[float] = None*, *metadata: Optional[Union[Mapping[str, Union[str, bytes]], Collection[Tuple[str, Union[str, bytes]]]]] = None*) → _RecvType

Coroutine to perform defined call.

Parameters

- **messages** – sequence of messages
- **timeout** (*float*) – request timeout (seconds)
- **metadata** – custom request metadata, dict or list of pairs

Returns
message

open(*, *timeout: Optional[float] = None*, *metadata: Optional[Union[Mapping[str, Union[str, bytes]], Collection[Tuple[str, Union[str, bytes]]]]] = None*) → *Stream*[_SendType, _RecvType]

Creates and returns *Stream* object to perform request to the server.

Nothing will happen to the current underlying HTTP/2 connection during this method call. It just initializes *Stream* object for you. Actual request will be sent only during *Stream.send_request()* or *Stream.send_message()* coroutine call.

Parameters

- **timeout** (*float*) – request timeout (seconds)
- **metadata** – custom request metadata, dict or list of pairs

Returns
Stream object

class grpclib.client.**StreamStreamMethod**(*channel: Channel*, *name: str*, *request_type: Type[_SendType]*, *reply_type: Type[_RecvType]*)

Represents STREAM-STREAM gRPC method type.

async **__call__**(*messages: Sequence[_SendType]*, ***, *timeout: Optional[float] = None*, *metadata: Optional[Union[Mapping[str, Union[str, bytes]], Collection[Tuple[str, Union[str, bytes]]]]] = None*) → List[_RecvType]

Coroutine to perform defined call.

Parameters

- **messages** – sequence of messages
- **timeout** (*float*) – request timeout (seconds)
- **metadata** – custom request metadata, dict or list of pairs

Returns
sequence of messages

open(***, *timeout: Optional[float] = None*, *metadata: Optional[Union[Mapping[str, Union[str, bytes]], Collection[Tuple[str, Union[str, bytes]]]]] = None*) → *Stream*[_SendType, _RecvType]

Creates and returns *Stream* object to perform request to the server.

Nothing will happen to the current underlying HTTP/2 connection during this method call. It just initializes *Stream* object for you. Actual request will be sent only during *Stream.send_request()* or *Stream.send_message()* coroutine call.

Parameters

- **timeout** (*float*) – request timeout (seconds)
- **metadata** – custom request metadata, dict or list of pairs

Returns
Stream object

4.4 Server

A single *Server* can serve arbitrary number of services:

```
server = Server([foo_svc, bar_svc, baz_svc])
```

To monitor health of your services you can use standard gRPC health checking protocol, details are here: *Health Checking*.

There is a special gRPC reflection protocol to inspect running servers and call their methods using command-line tools, details are here: *Reflection*. It is as simple as using curl.

It is also important to handle server's exit properly:

```
with graceful_exit([server]):
    await server.start(host, port)
    print(f'Serving on {host}:{port}')
    await server.wait_closed()
```

graceful_exit() helps you handle `SIGINT` and `SIGTERM` signals.

When things become complicated you can start using `AsyncExitStack` and `asynccontextmanager()` to manage lifecycle of your application and used resources:

```
async with AsyncExitStack() as stack:
    db = await stack.enter_async_context(setup_db())
    foo_svc = FooService(db)

    server = Server([foo_svc])
    stack.enter_context(graceful_exit([server]))
    await server.start(host, port)
    print(f'Serving on {host}:{port}')
    await server.wait_closed()
```

4.4.1 Reference

class grpclib.server.**Stream**(*stream: protocol.Stream*, *method_name: str*, *cardinality: Cardinality*, *recv_type: Type[_RecvType]*, *send_type: Type[_SendType]*, ***, *codec: CodecBase*, *status_details_codec: Optional[StatusDetailsCodecBase]*, *dispatch: _DispatchServerEvents*, *deadline: Optional[*Deadline*] = None*, *user_agent: Optional[str] = None*)

Represents gRPC method call – HTTP/2 request/stream, and everything you need to communicate with client in order to handle this request.

As you can see, every method handler accepts single positional argument - stream:

```
async def MakeLatte(self, stream: grpclib.server.Stream):
    task: cafe_pb2.LatteOrder = await stream.recv_message()
    ...
    await stream.send_message(empty_pb2.Empty())
```

This is true for every gRPC method type.

deadline

Deadline of the current request

metadata: Optional[_Metadata]

Invocation metadata, received with headers from the client. Represented as a multi-dict object.

user_agent

Client's user-agent

peer

Connection's peer info of type *Peer*

async recv_message() → Optional[_RecvType]

Coroutine to receive incoming message from the client.

If client sends UNARY request, then you can call this coroutine only once. If client sends STREAM request, then you should call this coroutine several times, until it returns None when the client has ended the stream. To simplify your code in this case, *Stream* class implements async iteration protocol, so you can use it like this:

```
async for message in stream:
    do_smth_with(message)
```

or even like this:

```
messages = [msg async for msg in stream]
```

HTTP/2 has flow control mechanism, so server will acknowledge received DATA frames as a message only after user consumes this coroutine.

Returns
message

async send_initial_metadata(*, *metadata: Optional[Union[Mapping[str, Union[str, bytes]], Collection[Tuple[str, Union[str, bytes]]]]] = None*) → None

Coroutine to send headers with initial metadata to the client.

In gRPC you can send initial metadata as soon as possible, because gRPC doesn't use *:status* pseudo header to indicate success or failure of the current request. gRPC uses trailers for this purpose, and trailers are sent during *send_trailing_metadata()* call, which should be called in the end.

Note: This coroutine will be called implicitly during first *send_message()* coroutine call, if not called before explicitly.

Parameters
metadata – custom initial metadata, dict or list of pairs

async send_message(*message: _SendType*) → None

Coroutine to send message to the client.

If server sends UNARY response, then you should call this coroutine only once. If server sends STREAM response, then you can call this coroutine as many times as you need.

Parameters
message – message object

async send_trailing_metadata(*, *status:* Status *= Status.OK*, *status_message: Optional[str] = None*, *status_details: Optional[Any] = None*, *metadata: Optional[Union[Mapping[str, Union[str, bytes]], Collection[Tuple[str, Union[str, bytes]]]]] = None*) → None

Coroutine to send trailers with trailing metadata to the client.

This coroutine allows sending trailers-only responses, in case of some failure conditions during handling current request, i.e. when `status is not OK`.

Note: This coroutine will be called implicitly at exit from request handler, with appropriate status code, if not called explicitly during handler execution.

Parameters

- **status** – resulting status of this coroutine call
- **status_message** – description for a status
- **metadata** – custom trailing metadata, dict or list of pairs

async cancel() → None

Coroutine to cancel this request/stream.

Server will send RST_STREAM frame to the client, so it will be explicitly informed that there is nothing to expect from the server regarding this request/stream.

class grpclib.server.**Server**(*handlers: Collection[IServable]*, *, *loop: Optional[AbstractEventLoop] = None*, *codec: Optional[CodecBase] = None*, *status_details_codec: Optional[StatusDetailsCodecBase] = None*, *config: Optional[*Configuration*] = None*)

HTTP/2 server, which uses gRPC service handlers to handle requests.

Handler is a subclass of the abstract base class, which was generated from .proto file:

```
class CoffeeMachine(cafe_grpc.CoffeeMachineBase):

    async def MakeLatte(self, stream):
        task: cafe_pb2.LatteOrder = await stream.recv_message()
        ...
        await stream.send_message(empty_pb2.Empty())

server = Server([CoffeeMachine()])
```

Parameters

- **handlers** – list of handlers
- **loop** – (deprecated) asyncio-compatible event loop
- **codec** – instance of a codec to encode and decode messages, if omitted `ProtoCodec` is used by default
- **status_details_codec** – instance of a status details codec to encode error details in a trailing metadata, if omitted `ProtoStatusDetailsCodec` is used by default

async start(*host: Optional[str] = None*, *port: Optional[int] = None*, *, *path: Optional[str] = None*, *family: socket.AddressFamily = AddressFamily.AF_UNSPEC*, *flags: socket.AddressInfo = AddressInfo.AI_PASSIVE*, *sock: Optional[socket] = None*, *backlog: int = 100*, *ssl: Optional[_ssl.SSLContext] = None*, *reuse_address: Optional[bool] = None*, *reuse_port: Optional[bool] = None*) → None

Coroutine to start the server.

Parameters

- **host** – can be a string, containing IPv4/v6 address or domain name. If host is None, server will be bound to all available interfaces.
- **port** – port number.
- **path** – UNIX domain socket path. If specified, host and port should be omitted (must be None).
- **family** – can be set to either `socket.AF_INET` or `socket.AF_INET6` to force the socket to use IPv4 or IPv6. If not set it will be determined from host.
- **flags** – is a bitmask for `getaddrinfo()`.
- **sock** – sock can optionally be specified in order to use a preexisting socket object. If specified, host and port should be omitted (must be None).

- **backlog** – is the maximum number of queued connections passed to listen().
- **ssl** – can be set to an SSLContext to enable SSL over the accepted connections.
- **reuse_address** – tells the kernel to reuse a local socket in TIME_WAIT state, without waiting for its natural timeout to expire.
- **reuse_port** – tells the kernel to allow this endpoint to be bound to the same port as other existing endpoints are bound to, so long as they all set this flag when being created.

close() → None

Stops accepting new connections, cancels all currently running requests. Request handlers are able to handle *CancelledError* and exit properly.

async wait_closed() → None

Coroutine to wait until all existing request handlers will exit properly.

grpclib.utils.**graceful_exit**(*servers: ~typing.Collection[IClosable], *, loop: ~typing.Optional[~asyncio.events.AbstractEventLoop] = None, signals: ~typing.Collection[int] = (<Signals.SIGINT: 2>, <Signals.SIGTERM: 15>)*) → Iterator[None]

Utility context-manager to help properly shutdown server in response to the OS signals

By default this context-manager handles `SIGINT` and `SIGTERM` signals.

There are two stages:

1. first received signal closes servers
2. subsequent signals raise `SystemExit` exception

Example:

```
async def main(...):
    ...
    with graceful_exit([server]):
        await server.start(host, port)
        print('Serving on {}:{}'.format(host, port))
        await server.wait_closed()
        print('Server closed')
```

First stage calls `server.close()` and `await server.wait_closed()` should complete successfully without errors. If server wasn't started yet, second stage runs to prevent server start.

Second stage raises `SystemExit` exception, but you will receive `asyncio.CancelledError` in your `async def main()` coroutine. You can use `try..finally` constructs and context-managers to properly handle this error.

This context-manager is designed to work in cooperation with `asyncio.run()` function:

```
if __name__ == '__main__':
    asyncio.run(main())
```

Parameters

- **servers** – list of servers
- **loop** – (deprecated) asyncio-compatible event loop
- **signals** – set of the OS signals to handle

Note: Not supported in Windows

4.5 Metadata

Structure of the gRPC call looks like this:

```
> :path /package/Method/
> ...
> ... request metadata

> data (request)

< :status 200
< ...
< ... initial metadata

< data (reply)

< grpc-status 0
< ...
< ... trailing metadata
```

The same as regular HTTP request but with trailers. So client can send request metadata and server can return initial and trailing metadata.

Metadata sent as regular HTTP headers. It may contain printable ascii text with spaces:

```
auth-token: 0d16ad85-6ce4-4773-a1be-9f62b2e886a3
```

Or it may contain binary data:

```
auth-token-bin: DRathWzkR3Ohvp9isuiGow
```

Binary metadata keys should contain `-bin` suffix and values should be encoded using base64 encoding without padding.

Keys with `grpc-` prefix are reserved for gRPC protocol. You can read more additional details here: gRPC Wire Format.

`grpclib` encodes and decodes binary metadata automatically. In Python you will receive text metadata as `str` type:

```
{"auth-token": "0d16ad85-6ce4-4773-a1be-9f62b2e886a3"}
```

Binary metadata you will receive as `bytes` type:

```
{"auth-token-bin": b"\r\x16\xad\x85l\xe4Gs\xa1\xbe\x9fb\xb2\xe8\x86\xa3"}
```

4.5.1 Client-Side

Sending metadata:

```
reply = await stub.Method(Request(), metadata={'auth-token': auth_token})
```

Sending and receiving metadata:

```
async with stub.Method.open(metadata={'auth-token': auth_token}) as stream:
    await stream.recv_initial_metadata()
    print(stream.initial_metadata)

    await stream.send_message(Request())
    reply = await stream.recv_message()

    await stream.recv_trailing_metadata()
    print(stream.trailing_metadata)
```

See reference docs for more details: *Client*.

4.5.2 Server-Side

Receiving and sending metadata:

```
class Service(ServiceBase):

    async def Method(self, stream):
        print(stream.metadata)  # request metadata

        await stream.send_initial_metadata(metadata={
            'begin-time': current_time(),
        })

        request = await stream.recv_message()
        ...
        await stream.send_message(Reply())

        await stream.send_trailing_metadata(metadata={
            'end-time': current_time(),
        })
```

See reference docs for more details: *Server*.

4.5.3 Reference

class grpclib.metadata.**Deadline**(*, *_timestamp: float*)

Represents request's deadline - fixed point in time

time_remaining() → float

Calculates remaining time for the current request completion

This function returns time in seconds as a floating point number, greater or equal to zero.

class grpclib.protocol.**Peer**(*transport: Transport*)

Represents an information about a connection's peer

addr() → Optional[Tuple[str, int]]

Returns the remote address to which we are connected

cert() → Optional[Dict[str, Any]]

Returns the peer certificate

Result of the `ssl.SSLSocket.getpeercert()`

4.6 Testing

You can use generated stubs to test your services. But it is not needed to setup connectivity over network interfaces. *grpclib* provides ability to use real client-side code, real server-side code, and real h2/gRPC protocol to test your services, with all the data sent in-memory.

4.6.1 Reference

class grpclib.testing.**ChannelFor**(*services: Collection[IServable]*, *codec: Optional[CodecBase] = None*, *status_details_codec: Optional[StatusDetailsCodecBase] = None*)

Manages specially initialised *Channel* with an in-memory transport to a *Server*

Example:

```
class Greeter(GreeterBase):
    ...

greeter = Greeter()

async with ChannelFor([greeter]) as channel:
    stub = GreeterStub(channel)
    response = await stub.SayHello(HelloRequest(name='Dr. Strange'))
    assert response.message == 'Hello, Dr. Strange!'
```

Parameters

- **services** – list of services you want to test
- **codec** – instance of a codec to encode and decode messages, if omitted `ProtoCodec` is used by default
- **status_details_codec** – instance of a status details codec to encode and decode error details in a trailing metadata, if omitted `ProtoStatusDetailsCodec` is used by default

4.7 Errors

GRPCError is a main error you should expect on the client-side and raise occasionally on the server-side.

4.7.1 Error Details

There is a possibility to send and receive rich error details, which may provide much more context than status and message alone. These details are encoded using `google.rpc.Status` message and sent with trailing metadata. This message becomes available after optional package install:

```
$ pip3 install googleapis-common-protos
```

There are some already defined error details in the `google.rpc.error_details_pb2` module, but you're not limited to them, you can send any message you want.

Here is how to send these details from the server-side:

```
from google.rpc.error_details_pb2 import BadRequest

async def Method(self, stream):
    ...
    raise GRPCError(
        Status.INVALID_ARGUMENT,
        'Request validation failed',
        [
            BadRequest(
                field_violations=[
                    BadRequest.FieldViolation(
                        field='title',
                        description='This field is required',
                    ),
                ],
            ),
        ],
    )
```

Here is how to dig into every detail on the client-side:

```
from google.rpc.error_details_pb2 import BadRequest

try:
    reply = await stub.Method(Request(...))
except GRPCError as err:
    if err.details:
        for detail in err.details:
            if isinstance(detail, BadRequest):
                for violation in detail.field_violations:
                    print(f'{violation.field}: {violation.description}')
```

Note: In order to automatically decode these messages (details), you have to import them, otherwise you will see such stubs in the list of error details:

```
Unknown('google.rpc.QuotaFailure')
```

4.7.2 Client-Side

Here is an example to illustrate how errors propagate from inside the grpclib methods back to the caller:

```
async with stub.SomeMethod.open() as stream:
    await stream.send_message(Request(...))
    reply = await stream.recv_message()  # gRPC error received during this call
```

Exceptions are propagated this way:

1. `CancelledError` is raised inside *`recv_message()`* coroutine to interrupt it
2. *`recv_message()`* coroutine handles this error and raise *`StreamTerminatedError`* instead or other error when it is possible to explain why coroutine was cancelled
3. when the `open()` context-manager exits, it may handle transitive errors such as *`StreamTerminatedError`* and raise proper *`GRPCError`* instead when possible

So here is a rule of thumb: expect *`GRPCError`* outside the `open()` context-manager:

```
try:
    async with stub.SomeMethod.open() as stream:
        await stream.send_message(Request(...))
        reply = await stream.recv_message()
except GRPCError as error:
    print(error.status, error.message)
```

4.7.3 Server-Side

Here is an example to illustrate how request cancellation is performed:

```
class Greeter(GreeterBase):
    async def SayHello(self, stream):
        try:
            ...
            await asyncio.sleep(1)  # cancel happens here
            ...
        finally:
            pass  # cleanup
```

1. Task running `SayHello` coroutine gets cancelled and `CancelledError` is raised inside it
2. You can use try..finally clause and/or context managers to properly cleanup used resources
3. When `SayHello` coroutine finishes, grpclib server internally re-raises `CancelledError` as `TimeoutError` or *`StreamTerminatedError`* to explain why request was cancelled
4. If cancellation isn't performed clearly, e.g. `SayHello` raises another exception instead of `CancelledError`, this error is logged.

4.7.4 Reference

exception grpclib.exceptions.**GRPCError**(*status:* Status, *message:* *Optional*[*str*] = *None*, *details:* *Optional*[*Any*] = *None*)

Expected error, may be raised during RPC call

There can be multiple origins of this error. It can be generated on the server-side and on the client-side. If this error originates from the server, on the wire this error is represented as `grpc-status` and `grpc-message` trailers. Possible values of the `grpc-status` trailer are described in the gRPC protocol definition. In `grpclib` these values are represented as *Status* enum.

Here are possible origins of this error:

- you may raise this error to cancel current call on the server-side or return non-OK *Status* using *send_trailing_metadata()* method *(e.g. resource not found)*
- server may return non-OK `grpc-status` in different failure conditions *(e.g. invalid request)*
- client raises this error for non-OK `grpc-status` from the server
- client may raise this error in different failure conditions *(e.g. server returned unsupported* `:content-type` *header)*

status

Status of the error

message

Error message

details

Error details

exception grpclib.exceptions.**ProtocolError**

Unexpected error, raised by `grpclib` when your code violates gRPC protocol

This error means that you probably should fix your code.

exception grpclib.exceptions.**StreamTerminatedError**

Unexpected error, raised when we receive `RST_STREAM` frame from the other side

This error means that the other side decided to forcefully cancel current call, probably because of a protocol error.

class grpclib.const.**Status**(*value*)

Predefined gRPC status codes represented as enum

See also: https://github.com/grpc/grpc/blob/master/doc/statuscodes.md

OK = 0

The operation completed successfully

CANCELLED = 1

The operation was cancelled (typically by the caller)

UNKNOWN = 2

Generic status to describe error when it can't be described using other statuses

INVALID_ARGUMENT = 3

Client specified an invalid argument

DEADLINE_EXCEEDED = 4

Deadline expired before operation could complete

NOT_FOUND = 5

Some requested entity was not found

ALREADY_EXISTS = 6

Some entity that we attempted to create already exists

PERMISSION_DENIED = 7

The caller does not have permission to execute the specified operation

RESOURCE_EXHAUSTED = 8

Some resource has been exhausted, perhaps a per-user quota, or perhaps the entire file system is out of space

FAILED_PRECONDITION = 9

Operation was rejected because the system is not in a state required for the operation's execution

ABORTED = 10

The operation was aborted

OUT_OF_RANGE = 11

Operation was attempted past the valid range

UNIMPLEMENTED = 12

Operation is not implemented or not supported/enabled in this service

INTERNAL = 13

Internal errors

UNAVAILABLE = 14

The service is currently unavailable

DATA_LOSS = 15

Unrecoverable data loss or corruption

UNAUTHENTICATED = 16

The request does not have valid authentication credentials for the operation

4.8 Configuration

Channel and *Server* classes accepts configuration via *Configuration* object to modify default behaviour.

Example:

```
from grpclib.config import Configuration

config = Configuration(
    http2_connection_window_size=2**20,  # 1 MiB
)
channel = Channel('localhost', 50051, config=config)
```

4.8.1 Reference

class grpclib.config.**Configuration**(*_keepalive_time: Union[float, NoneType] = <default>*, *_keepalive_timeout: float = 20.0*, *_keepalive_permit_without_calls: bool = False*, *_http2_max_pings_without_data: int = 2*, *_http2_min_sent_ping_interval_without_data: float = 300*, *http2_connection_window_size: int = 4194304*, *http2_stream_window_size: int = 4194304*)

http2_connection_window_size: int = 4194304

Sets inbound window size for a connection. HTTP/2 spec allows this value to be from 64 KiB to 2 GiB, 4 MiB is used by default

http2_stream_window_size: int = 4194304

Sets inbound window size for a stream. HTTP/2 spec allows this value to be from 64 KiB to 2 GiB, 4 MiB is used by default

4.9 Events

You can *listen()* for client-side events by using *Channel* instance as a target:

```
from grpclib.events import SendRequest

channel = Channel()

async def send_request(event: SendRequest):
    event.metadata['injected'] = 'successfully'

listen(channel, SendRequest, send_request)
```

For the server-side events you can *listen() Server* instance:

```
from grpclib.events import RecvRequest

server = Server([service])

async def recv_request(event: RecvRequest):
    print(event.metadata.get('injected'))

listen(server, RecvRequest, recv_request)
```

There are two types of event properties:

- **mutable**: you can change/mutate these properties and this will have an effect
- **read-only**: you can only read them

Listening callbacks are called in order: first added, first called. Each callback can `event.interrupt()` sequence of calls for a particular event:

```
async def authn_error(stream):
    raise GRPCError(Status.UNAUTHENTICATED)

async def recv_request(event: RecvRequest):
```

(continues on next page)

(continued from previous page)

```
    if event.metadata.get('auth-token') != SECRET:
        # provide custom RPC handler
        event.method_func = authn_error
        event.interrupt()

listen(server, RecvRequest, recv_request)
```

4.9.1 Common Events

grpclib.events.**listen**(*target: IEventsTarget*, *event_type: Type[_EventType]*, *callback: Callable[[_EventType], Coroutine[Any, Any, None]]*) → None

Registers a listener function for the given target and event type

```
async def callback(event: SomeEvent):
    print(event.data)

listen(target, SomeEvent, callback)
```

class grpclib.events.**SendMessage**(***kwargs*)

Dispatches before sending message to the other party

Parameters
message (*mutable*) – message to send

class grpclib.events.**RecvMessage**(***kwargs*)

Dispatches after message was received from the other party

Parameters
message (*mutable*) – received message

4.9.2 Client-Side Events

See also *SendMessage* and *RecvMessage*. You can listen for them on the client-side.

class grpclib.events.**SendRequest**(***kwargs*)

Dispatches before sending request to the server

Parameters

- **metadata** (*mutable*) – invocation metadata
- **method_name** (*read-only*) – RPC's method name
- **deadline** (*read-only*) – request's *Deadline*
- **content_type** (*read-only*) – request's content type

class grpclib.events.**RecvInitialMetadata**(***kwargs*)

Dispatches after headers with initial metadata were received from the server

Parameters
metadata (*mutable*) – initial metadata

class grpclib.events.**RecvTrailingMetadata**(***kwargs*)

Dispatches after trailers with trailing metadata were received from the server

Parameters

- **metadata** (*mutable*) – trailing metadata
- **status** (*read-only*) – status of the RPC call
- **status_message** (*read-only*) – description of the status
- **status_details** (*read-only*) – additional status details

4.9.3 Server-Side Events

See also *RecvMessage* and *SendMessage*. You can listen for them on the server-side.

class grpclib.events.**RecvRequest**(***kwargs*)

Dispatches after request was received from the client

Parameters

- **metadata** (*mutable*) – invocation metadata
- **method_func** (*mutable*) – coroutine function to process this request, accepts *Stream*
- **method_name** (*read-only*) – RPC's method name
- **deadline** (*read-only*) – request's *Deadline*
- **content_type** (*read-only*) – request's content type
- **user_agent** (*read-only*) – request's user agent
- **peer** (*read-only*) – request's *Peer*

class grpclib.events.**SendInitialMetadata**(***kwargs*)

Dispatches before sending headers with initial metadata to the client

Parameters
metadata (*mutable*) – initial metadata

class grpclib.events.**SendTrailingMetadata**(***kwargs*)

Dispatches before sending trailers with trailing metadata to the client

Parameters

- **metadata** (*mutable*) – trailing metadata
- **status** (*read-only*) – status of the RPC call
- **status_message** (*read-only*) – description of the status
- **status_details** (*read-only*) – additional status details

4.10 Encoding

GRPC supports sending messages using any encoding format, and grpclib supports this feature as well.

By default, gRPC interprets `application/grpc` content type as `application/grpc+proto` content type. So by default gRPC uses Protocol Buffers as encoding format.

But why content type has such name with a `proto` subtype? This is because messages in gRPC are sent as length-delimited stream of binary blobs. This format can't be changed, so content type should always be in the form of `application/grpc+{subtype}`, where `{subtype}` can be anything you want, e.g. `proto`, `fbs`, `json`, `thrift`, `bson`, `msgpack`.

4.10.1 Codec

In order to use custom serialization format, you should implement `CodecBase` abstract base class:

```
from grpclib.encoding.base import CodecBase

class JSONCodec(CodecBase):
    __content_subtype__ = 'json'

    def encode(self, message, message_type):
        return json.dumps(message, ensure_ascii=False).encode('utf-8')

    def decode(self, data: bytes, message_type):
        return json.loads(data.decode('utf-8'))
```

If your format doesn't have interface definition language (like protocol buffers language) and code-generation tools (like `protoc` compiler), you will have to manage your server-side and client-side code yourself. JSON format doesn't have such tools, so let's try define our server-side and client side code.

4.10.2 Naming Conventions

Even if you don't use Protocol Buffers for messages encoding, this language also defines coding style for services definition. These rules are related to service names and method names, which are used by gRPC to build `:path` pseudo header:

```
:path = /dotted.package.CamelCaseServiceName/CamelCaseMethodName
```

Protocol Buffers Style Guide says:

> You should use CamelCase (with an initial capital) for both the service name and any RPC method names.

4.10.3 Server example

```
from grpclib.const import Cardinality, Handler
from grpclib.server import Server

class PingServiceHandler:

    async def Ping(self, stream):
```

(continues on next page)

(continued from previous page)

```
        request = await stream.recv_message()
        ...
        await stream.send_message({'value': 'pong'})

    def __mapping__(self):
        return {
            '/ping.PingService/Ping': Handler(
                self.UnaryUnary,
                Cardinality.UNARY_UNARY,
                None,
                None,
            ),
        }

server = Server([PingServiceHandler()], codec=JSONCodec())
```

4.10.4 Client example

```
from grpclib.client import Channel, UnaryUnaryMethod

class PingServiceStub:

    def __init__(self, channel):
        self.Ping = UnaryUnaryMethod(
            channel,
            '/ping.PingService/Ping',
            None,
            None,
        )

channel = Channel(codec=JSONCodec())
ping_stub = PingServiceStub(channel)
...
await ping_stub.Ping({'value': 'ping'})
```

4.11 Health Checking

GRPC provides Health Checking Protocol to implement health checks. You can see it's latest definition here: grpc/health/v1/health.proto.

As you can see from the service definition, *Health* service should implement one or two methods: simple unary-unary `Check` method for synchronous checks and more sophisticated unary-stream `Watch` method to asynchronously wait for status changes. *grpclib* implements both of them.

grpclib also provides additional functionality to help write health checks, so users don't have to write a lot of code on their own. It is possible to implement health check in two ways (you can use both ways simultaneously):

- use *ServiceCheck* class by providing a callable object which can be called asynchronously to determine check's status
- use *ServiceStatus* class and change it's status by using *set* method

ServiceCheck is a simplest and most generic way to implement periodic checks.

ServiceStatus is for a more advanced usage, when you are able to detect and change check's status proactively (e.g. by detecting lost connection). And this way is more efficient and robust.

4.11.1 User Guide

Note: To test server's health we will use grpc_health_probe command.

Overall Server Health

The most simplest health checks:

```
from grpclib.health.service import Health

health = Health()

server = Server(handlers + [health])
```

Testing:

```
$ grpc_health_probe -addr=localhost:50051
healthy: SERVING
```

Overall server status is always `SERVING`.

If you want to add real checks:

```
from grpclib.health.service import Health, OVERALL

health = Health({OVERALL: [db_check, cache_check]})
```

Overall server status is `SERVING` if all checks are passing.

Detailed Services Health

If you want to provide different checks for different services:

```
foo = FooService()
bar = BarService()

health = Health({
    foo: [a_check, b_check],
    bar: [b_check, c_check],
})
```

Testing:

```
$ grpc_health_probe -addr=localhost:50051 -service acme.FooService
healthy: SERVING
$ grpc_health_probe -addr=localhost:50051 -service acme.BarService
```

(continues on next page)

(continued from previous page)

```
healthy: NOT_SERVING
$ grpc_health_probe -addr=localhost:50051
healthy: NOT_SERVING
```

- `acme.FooService` is healthy if `a_check` and `b_check` are passing
- `acme.BarService` is healthy if `b_check` and `c_check` are passing
- Overall health status depends on all checks

You can also override checks list for overall server's health status:

```
foo = FooService()
bar = BarService()

health = Health({
    foo: [a_check, b_check],
    bar: [b_check, c_check],
    OVERALL: [a_check, c_check],
})
```

4.11.2 Reference

grpclib.health.service.**OVERALL = <grpclib.health.service._Overall object>**

Represents overall health status of all services

class grpclib.health.service.**Health**(*checks: Optional[Mapping[ICheckable, Collection[CheckBase]]] = None*)

Health-checking service

Example:

```
from grpclib.health.service import Health

auth = AuthService()
billing = BillingService()

health = Health({
    auth: [redis_status],
    billing: [db_check],
})

server = Server([auth, billing, health])
```

async Check(*stream:* Stream*[HealthCheckRequest, HealthCheckResponse]*) → None

Implements synchronous periodic checks

class grpclib.health.check.**ServiceCheck**(*func: Callable[[], Awaitable[Optional[bool]]], *, loop: Optional[AbstractEventLoop] = None, check_ttl: float = 30, check_timeout: float = 10*)

Performs periodic checks

Example:

```
async def db_test():
    # raised exceptions are the same as returning False,
    # except that exceptions will be logged
    await db.execute('SELECT 1;')
    return True

db_check = ServiceCheck(db_test)
```

Parameters

- **func** – callable object which returns awaitable object, where result is one of: `True` (healthy), `False` (unhealthy), or `None` (unknown)
- **loop** – (deprecated) asyncio-compatible event loop
- **check_ttl** – how long we can cache result of the previous check
- **check_timeout** – timeout for this check

class grpclib.health.check.**ServiceStatus**(*, *loop: Optional[AbstractEventLoop] = None*)

Contains status of a proactive check

Example:

```
redis_status = ServiceStatus()

# detected that Redis is available
redis_status.set(True)

# detected that Redis is unavailable
redis_status.set(False)
```

Parameters

loop – (deprecated) asyncio-compatible event loop

set(*value: Optional[bool]*) → None

Sets current status of a check

Parameters

value – `True` (healthy), `False` (unhealthy), or `None` (unknown)

4.12 Reflection

Server reflection is an optional extension, which describes services, implemented on the server.

In examples we will use `grpc_cli` command-line tool and `helloworld` example. We will use *`extend()`* method to add server reflection.

Then we will be able to…

List services on the server:

```
$ grpc_cli ls localhost:50051
helloworld.Greeter
```

List methods of the service:

```
$ grpc_cli ls localhost:50051 helloworld.Greeter -l
filename: helloworld/helloworld.proto
package: helloworld;
service Greeter {
  rpc SayHello(helloworld.HelloRequest) returns (helloworld.HelloReply) {}
}
```

Describe messages:

```
$ grpc_cli type localhost:50051 helloworld.HelloRequest
message HelloRequest {
  string name = 1;
}
```

Call simple methods:

```
$ grpc_cli call localhost:50051 helloworld.Greeter.SayHello "name: 'Dr. Strange'"
connecting to localhost:50051
message: "Hello, Dr. Strange!"

Rpc succeeded with OK status
```

And all of these done without downloading .proto files and compiling them into other source files in order to create stubs.

4.12.1 Reference

class grpclib.reflection.service.**ServerReflection**(*, *_service_names: Collection[str]*)

Implements server reflection protocol.

classmethod extend(*services: Collection[IServable]*) → List[IServable]

Extends services list with reflection service:

```
from grpclib.reflection.service import ServerReflection

services = [Greeter()]
services = ServerReflection.extend(services)

server = Server(services)
...
```

Returns new services list with reflection support added.

PYTHON MODULE INDEX

g

INDEX

Symbols

A

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W