sliplib

Release 0.6.1

Ruud de Jong

May 22, 2020

CONTENTS

The *sliplib* module implements the encoding and decoding functionality for SLIP packets, as described in **RFC 1055**. It defines encoding, decoding, and validation functions, as well as a driver class that can be used to implement a SLIP protocol stack, and higher-level classes that apply the SLIP protocol to TCP connections or IO streams. Read the documentation for detailed information.

CHAPTER

ONE

BACKGROUND

The SLIP protocol is described in **RFC 1055** (*A Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP*, J. Romkey, June 1988). The original purpose of the protocol is to provide a mechanism to indicate the boundaries of IP packets, in particular when the IP packets are sent over a connection that does not provide a framing mechanism, such as serial lines or dial-up connections.

There is, however, nothing specific to IP in the SLIP protocol. SLIP offers a generic framing method that can be used for any type of data that must be transmitted over a (continuous) byte stream. In fact, the main reason for creating this module was the need to communicate with a third-party application that used SLIP over TCP (which is a continuous byte stream) to frame variable length data structures.

CHAPTER

TWO

USAGE

2.1 Low-level usage

The recommended basic usage is to run all encoding and decoding operations through an instantiation *driver* of the *Driver* class, in combination with the appropriate I/O code. The *Driver* class itself works without any I/O, and can therefore be used with any networking code, or any bytestream like pipes, serial I/O, etc. It can work in synchronous as well as in asynchronous environments.

The *Driver* class offers the methods *send* and *receive* to handle the conversion between messages and SLIP-encoded packets.

2.2 High-level usage

The module also provides a *SlipWrapper* abstract baseclass that provides the methods *send_msg* and *recv_msg* to send and receive single SLIP-encoded messages. This base class wraps an instance of the *Driver* class with a user-provided stream.

Two concrete subclasses of *SlipWrapper* are provided:

- *SlipStream* allows the wrapping of a byte IO stream.
- *SlipSocket* allows the wrapping of a TCP socket.

In addition, the module also provides a *SlipRequestHandler* to facilitate the creation of TCP servers that can handle SLIP-encoded messages.

CHAPTER

THREE

ERROR HANDLING

Contrary to the reference implementation described in **RFC 1055**, which chooses to essentially ignore protocol errors, the functions and classes in the *sliplib* module uses a *ProtocolError* exception to indicate protocol errors, i.e. SLIP packets with invalid byte sequences. The *Driver* class raises the *ProtocolError* exception as soon as a SLIP packet with an invalid byte sequence is received . The *SlipWrapper* class and its subclasses catch the *ProtocolError*s raised by the *Driver* class, and re-raise them when an attempt is made to read the contents of a SLIP packet with invalid data.

CHAPTER

FOUR

CHANGELOG

4.1 v0.6.0

- Added support for unbuffered byte streams in SlipStream (issue #16).
- Deprecated direct access to wrapped bytestream (SlipStream) and socket (SlipSocket)
- Updated documentation and examples

4.2 v0.5.0

- Made SlipWrapper and its derived classes iterable (issue #18).

4.3 v0.4.0

- Removed sphinx as install dependency (issue #9). Sphinx is only required for documentation development.
- Changes in automated testing:
  - Added testing against Python 3.8.
  - Added macOS testing.
  - Removed testing against Python 3.4.

4.4 v0.3.0

- First general available beta release.

4.4.1 Module Contents

Introduction

The *sliplib* module implements the encoding and decoding functionality for SLIP packets, as described in **RFC 1055**. It defines encoding, decoding, and validation functions, as well as various classes that can be used to to wrap the SLIP protocol over different kinds of byte streams.

The SLIP protocol is described in **RFC 1055** (*A Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP*, J. Romkey, June 1988). The original purpose of the protocol is to provide a mechanism to indicate the boundaries of IP packets, in particular when the IP packets are sent over a connection that does not provide a framing mechanism, such as serial lines or dial-up connections.

There is, however, nothing specific to IP in the SLIP protocol. The protocol describes a generic framing method that can be used for any type of data that must be transmitted over a (continuous) byte stream. In fact, the main reason for creating this module was the need to communicate with a third-party application that used SLIP over TCP (which is a continuous byte stream) to frame variable length data structures.

The SLIP protocol uses four special byte values:

Byte value	Name	Purpose
`0xc0`	`END`	to delimit messages
`0xdb`	`ESC`	to escape `END` or `ESC` bytes in the message
`0xdc`	`ESC_END`	the escaped value of the `END` byte
`0xdd`	`ESC_ESC`	the escaped value of the `ESC` byte

An `END` byte in the message is encoded as the sequence `ESC+ESC_END` (`b'\xdb\xdc'`) in the slip packet, and an `ESC` byte in the message is encoded as the sequence `ESC+ESC_ESC` (`b'\xdb\xdd'`).

Decoded	Encoded
`b'\xc0'`	`b'\xdb\xdc'`
`b'\xdb'`	`b'\xdb\xdd'`

As a consequence, an `ESC` byte in an encoded SLIP packet must always be followed by an `ESC_END` or an `ESC_ESC` byte; anything else is a protocol error.

Low level Usage

Constants

END

ESC

ESC_END

ESC_ESC

These constants represent the special bytes used by SLIP for delimiting and encoding messages.

Functions

The following are lower-level functions, that should normally not be used directly.

encode(*msg*)
Encodes a message (a byte sequence) into a SLIP-encoded packet.

Parameters **msg** (*bytes*) – The message that must be encoded

Returns The SLIP-encoded message

Return type bytes

decode(*packet*)
Retrieves the message from the SLIP-encoded packet.

Parameters **packet** (*bytes*) – The SLIP-encoded message. Note that this must be exactly one complete packet. The *decode()* function does not provide any buffering for incomplete packages, nor does it provide support for decoding data with multiple packets.

Returns The decoded message

Raises ***ProtocolError*** – if the packet contains an invalid byte sequence.

Return type bytes

is_valid(*packet*)
Indicates if the packet's contents conform to the SLIP specification.

A packet is valid if:

- It contains no *END* bytes other than leading and/or trailing *END* bytes, and
- Each *ESC* byte is followed by either an *ESC_END* or an *ESC_ESC* byte.

Parameters **packet** (*bytes*) – The packet to inspect.

Returns `True` if the packet is valid, `False` otherwise

Return type bool

Classes

class Driver
Class to handle the SLIP-encoding and decoding of messages

This class manages the handling of encoding and decoding of messages according to the SLIP protocol.

Class *Driver* offers the following methods:

Return type None

send(*message*)
Encodes a message into a SLIP-encoded packet.

The message can be any arbitrary byte sequence.

Parameters **message** (*bytes*) – The message that must be encoded.

Returns A packet with the SLIP-encoded message.

Return type bytes

receive(*data*)
Decodes data and gives a list of decoded messages.

Processes `data`, which must be a bytes-like object, and returns a (possibly empty) list with `bytes` objects, each containing a decoded message. Any non-terminated SLIP packets in `data` are buffered, and processed with the next call to *receive()*.

Parameters **data** (*Union[bytes, int]*) – A bytes-like object to be processed.

An empty `data` parameter forces the internal buffer to be flushed and decoded.

To accommodate iteration over byte sequences, an integer in the range(0, 256) is also accepted.

Returns A (possibly empty) list of decoded messages.

Raises ***ProtocolError*** – When *data* contains an invalid byte sequence.

Return type List[bytes]

To enable recovery from a `ProtocolError`, the *Driver* class offers the following attribute and method:

messages
A list of decoded messages.

The read-only attribute *messages* contains the messages that were already decoded before a `ProtocolError` was raised. This enables the handler of the `ProtocolError` exception to recover the messages up to the point where the error occurred. This attribute is cleared after it has been read.

flush()
Gives a list of decoded messages.

Decodes the packets in the internal buffer. This enables the continuation of the processing of received packets after a `ProtocolError` has been handled.

Returns A (possibly empty) list of decoded messages from the buffered packets.

Raises ***ProtocolError*** – When any of the buffered packets contains an invalid byte sequence.

Return type List[bytes]

High Level Usage

SlipWrapper

class SlipWrapper(*stream*)
Base class that provides a message based interface to a byte stream

SlipWrapper combines a `Driver` instance with a byte stream. The *SlipWrapper* class is an abstract base class. It offers the methods *send_msg()* and *recv_msg()* to send and receive single messages over the byte stream, but it does not of itself provide the means to interact with the stream.

To interact with a concrete stream, a derived class must implement the methods *send_bytes()* and *recv_bytes()* to write to and read from the stream.

A *SlipWrapper* instance can be iterated over. Each iteration will provide the next message that is received from the byte stream.

Changed in version 0.5: Allow iteration over a *SlipWrapper* instance.

To instantiate a *SlipWrapper*, the user must provide an existing byte stream

Parameters **stream** (*bytestream*) – The byte stream that will be wrapped.

Class *SlipWrapper* offers the following methods and attributes:

send_msg(*message*)
Send a SLIP-encoded message over the stream.

Parameters **message** (*bytes*) – The message to encode and send

Return type None

recv_msg()
Receive a single message from the stream.

Returns A SLIP-decoded message

Return type bytes

Raises ***ProtocolError*** – when a SLIP protocol error has been encountered. A subsequent call to *recv_msg()* (after handling the exception) will return the message from the next packet.

driver
The *SlipWrapper*'s `Driver` instance.

stream
The wrapped *stream*.

In addition, *SlipWrapper* requires that derived classes implement the following methods:

send_bytes(*packet*)
Send a packet over the stream.

Derived classes must implement this method.

Parameters **packet** (*bytes*) – the packet to send over the stream

Return type None

recv_bytes()
Receive data from the stream.

Derived classes must implement this method.

Note: The convention used within the *SlipWrapper* class is that *recv_bytes()* returns an empty bytes object to indicate that the end of the byte stream has been reached and no further data will be received. Derived implementations must ensure that this convention is followed.

Returns The bytes received from the stream

Return type bytes

SlipStream

class SlipStream(*stream*[, *chunk_size*])

Bases: *sliplib.slipwrapper.SlipWrapper*

Class that wraps an IO stream with a `Driver`

SlipStream combines a `Driver` instance with a concrete byte stream. The byte stream must support the methods `read()` and `write()`. To avoid conflicts and ambiguities caused by different *newline* conventions, streams that have an `encoding` attribute (such as `io.StringIO` objects, or text files that are not opened in binary mode) are not accepted as a byte stream.

The *SlipStream* class has all the methods and attributes from its base class `SlipWrapper`. In addition it directly exposes all methods and attributes of the contained `stream`, except for the following:

- `read*()` and `write*()`. These methods are not supported, because byte-oriented read and write operations would invalidate the internal state maintained by *SlipStream*.
- Similarly, `seek()`, `tell()`, and `truncate()` are not supported, because repositioning or truncating the stream would invalidate the internal state.
- `raw()`, `detach()` and other methods that provide access to or manipulate the stream's internal data.

In stead of the `read*()` and `write*()` methods a *SlipStream* object provides the method `recv_msg()` and `send_msg()` to read and write SLIP-encoded messages.

Deprecated since version 0.6: Direct access to the methods and attributes of the contained `stream` will be removed in version 1.0

To instantiate a *SlipStream* object, the user must provide a pre-constructed open byte stream that is ready for reading and/or writing

Parameters

- **stream** (*bytestream*) – The byte stream that will be wrapped.
- **chunk_size** (*int*) – the number of bytes to read per read operation. The default value for *chunck_size* is *io.DEFAULT_BUFFER_SIZE*. Setting the *chunk_size* is useful when the stream has a low bandwidth and/or bursty data (e.g. a serial port interface). In such cases it is useful to have a *chunk_size* of 1, to avoid that the application hangs or becomes unresponsive.

New in version 0.6: The *chunk_size* parameter.

A *SlipStream* instance can e.g. be useful to read slip-encoded messages from a file:

```
with open('/path/to/a/slip/encoded/file', mode='rb') as f:
    slip_file = SlipStream(f)
    for msg in slip_file:
        # Do something with the message
```

A *SlipStream* instance has the following attributes in addition to the attributes offered by its base class `SlipWrapper`:

readable
Indicates if the wrapped stream is readable. The value is *True* if the readability of the wrapped stream cannot be determined.

writable
Indicates if the wrapped stream is writable. The value is *True* if the writabilty of the wrapped stream cannot be determined.

SlipSocket

class SlipSocket(*sock*)

Bases: *sliplib.slipwrapper.SlipWrapper*

Class that wraps a TCP *socket* with a `Driver`

SlipSocket combines a `Driver` instance with a *socket*. The `SlipStream` class has all the methods from its base class `SlipWrapper`. In addition it directly exposes all methods and attributes of the contained *socket*, except for the following:

- `send*()` and `recv*()`. These methods are not supported, because byte-oriented send and receive operations would invalidate the internal state maintained by *SlipSocket*.
- Similarly, `makefile()` is not supported, because byte- or line-oriented read and write operations would invalidate the internal state.
- `share()` (Windows only) and `dup()`. The internal state of the *SlipSocket* would have to be duplicated and shared to make these methods meaningful. Because of the lack of a convincing use case for this, sharing and duplication is not supported.
- The *accept()* method is delegated to the contained *socket*, but the socket that is returned by the *socket*'s *accept()* method is automatically wrapped in a *SlipSocket* object.

In stead of the *socket*'s `send*()` and `recv*()` methods a *SlipSocket* provides the method `send_msg()` and `recv_msg()` to send and receive SLIP-encoded messages.

Deprecated since version 0.6: Direct access to the methods and attributes of the contained *socket* other than *family*, *type*, and *proto* will be removed in version 1.0

Only TCP sockets are supported. Using the SLIP protocol on UDP sockets is not supported for the following reasons:

- UDP is datagram-based. Using SLIP with UDP therefore introduces ambiguity: should SLIP packets be allowed to span multiple UDP datagrams or not?
- UDP does not guarantee delivery, and does not guarantee that datagrams are delivered in the correct order.

To instantiate a *SlipSocket*, the user must provide a pre-constructed TCP *socket*. An alternative way to instantiate s SlipSocket is to use the class method *create_connection()*.

Parameters **sock** (*socket.socket*) – An existing TCP socket, i.e. a socket with type `socket.SOCK_STREAM`

Class *SlipSocket* offers the following methods in addition to the methods offered by its base class `SlipWrapper`:

accept()

Accepts an incoming connection.

Returns A (*SlipSocket*, remote_address) pair. The *SlipSocket* object can be used to exchange SLIP-encoded data with the socket at the *remote_address*.

Return type Tuple[*SlipSocket*, Tuple]

See also:

`socket.socket.accept()`

classmethod create_connection(*address*, *timeout=None*, *source_address=None*)

Create a SlipSocket connection.

This convenience method creates a connection to a socket at the specified address using the socket.create_connection() function. The socket that is returned from that call is automatically wrapped in a *SlipSocket* object.

Parameters

- **address** (*Address*) – The remote address.
- **timeout** (*float*) – Optional timeout value.
- **source_address** (*Address*) – Optional local address for the near socket.

Returns A *SlipSocket* that is connected to the socket at the remote address.

Return type *SlipSocket*

See also:

socket.create_connection()

Note: The *accept()* and *create_connection()* methods do not magically turn the socket at the remote address into a SlipSocket. For the connection to work properly, the remote socket must already have been configured to use the SLIP protocol.

The following commonly used socket.socket methods are exposed through a *SlipSocket* object. These methods are simply delegated to the wrapped *socket* instance.

bind(*address*)

Bind the *SlipSocket* to *address*.

Parameters **address** (*Tuple*) – The IP address to bind to.

Return type None

See also:

socket.socket.bind()

close()

Close the *SlipSocket*.

See also:

socket.socket.close()

Return type None

connect(*address*)

Connect *SlipSocket* to a remote socket at *address*.

Parameters **address** (*Tuple*) – The IP address of the remote socket.

Return type None

See also:

socket.socket.connect()

connect_ex(*address*)

Connect *SlipSocket* to a remote socket at *address*.

Parameters **address** (*Tuple*) – The IP address of the remote socket.

Return type None

See also:

`socket.socket.connect_ex()`

getpeername()
Get the IP address of the remote socket to which *SlipSocket* is connected.

Returns The remote IP address.

Return type Tuple

See also:

`socket.socket.getpeername()`

getsockname()
Get *SlipSocket*'s own address.

Returns The local IP address.

Return type Tuple

See also:

`socket.socket.getsockname()`

listen([*backlog*])
Enable a *SlipSocket* server to accept connections.

Parameters **backlog** (*int*) – The maximum number of waiting connections.

Return type None

See also:

`socket.socket.listen()`

shutdown(*how*)
Shutdown the connection.

Parameters **how** (*int*) – Flag to indicate which halves of the connection must be shut down.

Return type None

See also:

`socket.socket.shutdown()`

Since the wrapped socket is available as the *socket* attribute, any other `socket.socket` method can be invoked through that attribute.

Warning: Avoid using `socket.socket` methods that affect the bytes that are sent or received through the socket. Doing so will invalidate the internal state of the enclosed `Driver` instance, resulting in corrupted SLIP messages. In particular, do not use any of the `recv*()` or `send*()` methods on the *socket* attribute.

A *SlipSocket* instance has the following attributes in addition to the attributes offered by its base class `SlipWrapper`:

socket
The wrapped *socket*. This is actually just an alias for the `stream` attribute in the base class.

family
The wrapped socket's address family. Usually `socket.AF_INET` (IPv4) or `socket.AF_INET6` (IPv6).

type
The wrapped socket's type. Always `socket.SOCK_STREAM`.

proto
The wrapped socket's protocol number. Usually 0.

SlipRequestHandler

class SlipRequestHandler(*request*, *client_address*, *server*)
Bases: `socketserver.BaseRequestHandler`

Base class for request handlers for SLIP-based communication

This class is derived from `socketserver.BaseRequestHandler` for the purpose of creating TCP server instances that can handle incoming SLIP-based connections.

To implement a specific behaviour, all that is needed is to derive a class that defines a *`handle()`* method that uses `self.request` to send and receive SLIP-encoded messages.

The interface is identical to that offered by the `socketserver.BaseRequestHandler` baseclass. To do anything useful, a derived class must define a new *`handle()`* method, and may override any of the other methods.

setup()
Initializes the request handler.

The original socket (available via `self.request`) is wrapped in a `SlipSocket` object. Derived classes may override this method, but should call `super().setup()` before accessing any `SlipSocket` methods or attributes on `self.request`.

Return type None

handle()
Services the request. Must be defined by a derived class.

Note that in general it does not make sense to use a *`SlipRequestHandler`* object to handle a single transmission, as is e.g. common with HTTP. The purpose of the SLIP protocol is to allow separation of messages in a continuous byte stream. As such, it is expected that the *`handle()`* method of a derived class is capable of handling multiple SLIP messages:

```
def handle(self):
    while True:
        msg = self.request.recv_msg()
        if msg == b'':
            break
        # Do something with the message
```

Return type None

finish()
Performs any cleanup actions.

The default implementation does nothing.

Return type None

Exceptions

exception ProtocolError

Exception to indicate that a SLIP protocol error has occurred.

This exception is raised when an attempt is made to decode a packet with an invalid byte sequence. An invalid byte sequence is either an `ESC` byte followed by any byte that is not an `ESC_ESC` or `ESC_END` byte, or a trailing `ESC` byte as last byte of the packet.

The *`ProtocolError`* carries the invalid packet as the first (and only) element in in its `args` tuple.

4.4.2 Examples

The directory `examples` in SlipLib's GitHub repository contains some basic examples on how the *`sliplib`* module can be used.

Echoserver

This directory contains an example server and client application that demonstrate a basic use-case for Slip-encoded messages. The example works both for IPv4 and IPv6 sockets.

Server

The `server.py` example file is a demonstrator echo server. It uses a subclass of `SlipRequestHandler` that transforms the `request` attribute into a dedicated socket subclass that prints the raw data that is received and sent. The request handler prints the decoded message, and then reverses the order of the bytes in the encoded message (so `abc` becomes `cab`), and sends it back to the client.

Client

The `client.py` example file is a client for the demonstrator echo server . It prompts the user for a message, encodes it in a Slip packet, sends it to the server, and prints the decoded reply it gets back from the server. This is repeated until the user enters an empty message.

Usage

Open a terminal window in the `echoserver` directory and run the `server_ipv6.py` script. This will start the server and print the address on which the server is listening.

```
$ python server.py
Slip server listening on localhost, port 59454
```

Then in another terminal window in the same directory run the `client.py` script with the port number reported by the server.

```
$ python client.py 59454
Connecting to server on port 59454
Connected to ('127.0.0.1', 59454)
Message>
```

You can now enter a message, and the client will print the response from the server before prompting for the next message. An empty message stops both the client and the server.

```
$ python client.py 59454
Connecting to server on port 59454
Connected to ('127.0.0.1', 59454)
Message> hallo
Response: b'ollah'
Message> bye
Response: b'eyb'
Message>
$
```

The server will have printed the following information:

```
$ python server_ipv6.py
Slip server listening on localhost, port 59454
Incoming connection from ('127.0.0.1', 59458)
Raw data received: b'\xc0hallo\xc0'
Decoded data: b'hallo'
Sending raw data: b'\xc0ollah\xc0'
Raw data received: b'\xc0bye\xc0'
Decoded data: b'bye'
Sending raw data: b'\xc0eyb\xc0'
Raw data received: b''
Decoded data: b''
Closing down
$
```

Running on IPv6

By running the server with the argument `ipv6`, an IPv6-based connection will be established.

In the server terminal window:

```
$ python server.py ipv6
Slip server listening on localhost, port 59454
Incoming connection from ('::1', 59458, 0, 0)
...
```

In the client terminal window:

```
$ python client.py 59454
Connecting to server on port 59454
Connected to ('::1', 59454, 0, 0)
Message>
...
```

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

s

INDEX

T

W