Jeepney Documentation

Release 0.8.0

Thomas Kluyver

Dec 19, 2022

Contents

1 Connecting to DBus and sending messages **3**
- 1.1 Connections and Routers . . . 4
- 1.2 Message generators and proxies . . . 5
- 1.3 Sending & receiving file descriptors . . . 6

2 Making and parsing messages **7**
- 2.1 Making messages . . . 7

3 Generating D-Bus wrappers **9**

4 API reference **11**
- 4.1 Core API . . . 11
- 4.2 Common messages . . . 15
- 4.3 Authentication . . . 16
- 4.4 File descriptor support . . . 18
- 4.5 Blocking I/O . . . 19
- 4.6 Blocking I/O with threads . . . 20
- 4.7 Trio integration . . . 22
- 4.8 Asyncio integration . . . 24
- 4.9 I/O Exceptions . . . 25

5 Design & Limitations **27**
- 5.1 Non-goals . . . 27
- 5.2 Alternatives . . . 28

6 What is D-Bus? **29**
- 6.1 Methods & signals . . . 30
- 6.2 Names . . . 30
- 6.3 Message buses . . . 30
- 6.4 Special features . . . 31

7 Release notes **33**
- 7.1 0.8 . . . 33
- 7.2 0.7.1 . . . 33
- 7.3 0.7 . . . 33
- 7.4 0.6 . . . 34
- 7.5 0.5 . . . 34

7.6 0.4.3 . . . 34
7.7 0.4.2 . . . 34
7.8 0.4.1 . . . 35
7.9 0.4 . . . 35

8 Indices and tables **37**

Python Module Index **39**

Index **41**

Jeepney is a pure Python interface to D-Bus, a protocol for interprocess communication on desktop Linux (mostly). See *What is D-Bus?* for more background on what it can do.

The core of Jeepney is I/O free, and the `jeepney.io` package contains bindings for different event loops to handle I/O. Jeepney tries to be *non-magical*, so you may have to write a bit more code than with other interfaces such as dbus-python or pydbus.

Jeepney doesn't rely on libdbus or other compiled libraries, so it's easy to install with Python tools like `pip`:

```
pip install jeepney
```

For most use cases, the D-Bus daemon needs to be running on your computer; this is a standard part of most modern Linux desktops.

Contents:

CHAPTER 1

Connecting to DBus and sending messages

Jeepney can be used with several different frameworks:

- Blocking I/O
- Multi-threading with the threading module
- Trio
- asyncio

For each of these, there is a module in `jeepney.io` providing the integration layer.

Here's an example of sending a desktop notification, using blocking I/O:

```
from jeepney import DBusAddress, new_method_call
from jeepney.io.blocking import open_dbus_connection

notifications = DBusAddress('/org/freedesktop/Notifications',
                            bus_name='org.freedesktop.Notifications',
                            interface='org.freedesktop.Notifications')

connection = open_dbus_connection(bus='SESSION')

# Construct a new D-Bus message. new_method_call takes the address, the
# method name, the signature string, and a tuple of arguments.
msg = new_method_call(notifications, 'Notify', 'susssasa{sv}i',
                      ('jeepney_test',  # App name
                       0,    # Not replacing any previous notification
                       '',   # Icon
                       'Hello, world!',  # Summary
                       'This is an example notification from Jeepney',
                       [], {},  # Actions, hints
                       -1,      # expire_timeout (-1 = default)
                      ))

# Send the message and wait for the reply
```

(continues on next page)

(continued from previous page)

```
reply = connection.send_and_get_reply(msg)
print('Notification ID:', reply.body[0])

connection.close()
```

And here is the same thing using asyncio:

```
"""Send a desktop notification

See also aio_notify.py, which does the same with the higher-level Proxy API.
"""
import asyncio

from jeepney import DBusAddress, new_method_call
from jeepney.io.asyncio import open_dbus_router

notifications = DBusAddress('/org/freedesktop/Notifications',
                            bus_name='org.freedesktop.Notifications',
                            interface='org.freedesktop.Notifications')

async def send_notification():
    msg = new_method_call(notifications, 'Notify', 'susssasa{sv}i',
                          ('jeepney_test',  # App name
                           0,      # Not replacing any previous notification
                           '',     # Icon
                           'Hello, world!',  # Summary
                           'This is an example notification from Jeepney',
                           [], {},  # Actions, hints
                           -1,      # expire_timeout (-1 = default)
                          ))
    # Send the message and await the reply
    async with open_dbus_router() as router:
        reply = await router.send_and_get_reply(msg)
        print('Notification ID:', reply.body[0])

loop = asyncio.get_event_loop()
loop.run_until_complete(send_notification())
```

See the examples folder in Jeepney's source repository for more examples.

1.1 Connections and Routers

Each integration (except blocking I/O) can create *connections* and *routers*.

Routers are useful for calling methods in other processes. Routers let you send a request and wait for a reply, using a *proxy* or with `router.send_and_get_reply()`. You can also filter incoming messages into queues, e.g. to wait for a specific signal. But if messages arrive faster than they are processed, these queues fill up, and messages may be dropped.

Connections are simpler: they let you send and receive messages, but `conn.receive()` will give you the next message read, whatever that is. You'd use this to write a server which responds to incoming messages. A connection will never discard an incoming message.

Note: For blocking, single-threaded I/O, the connection doubles as a router. Incoming messages while you're waiting for a reply will be filtered, and you can also filter the next message by calling `conn.recv_messages()`.

Routers for the other integrations receive messages in a background task.

1.2 Message generators and proxies

If you're calling a number of different methods, you can make a *message generator* class containing their definitions. Jeepney includes a tool to generate these classes automatically—see *Generating D-Bus wrappers*.

Message generators define how to construct messages. *Proxies* are wrappers around message generators which send a message and get the reply back.

Let's rewrite the example above to use a message generator and a proxy:

```
"""Send a desktop notification

See also aio_notify_noproxy.py, which does the same with lower-level APIs
"""
import asyncio

from jeepney import MessageGenerator, new_method_call
from jeepney.io.asyncio import open_dbus_router, Proxy

# ---- Message generator, created by jeepney.bindgen ----
class Notifications(MessageGenerator):
    interface = 'org.freedesktop.Notifications'

    def __init__(self, object_path='/org/freedesktop/Notifications',
                 bus_name='org.freedesktop.Notifications'):
        super().__init__(object_path=object_path, bus_name=bus_name)

    def Notify(self, arg_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6, arg_7):
        return new_method_call(self, 'Notify', 'susssasa{sv}i',
                               (arg_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6, arg_
↪7))

    def CloseNotification(self, arg_0):
        return new_method_call(self, 'CloseNotification', 'u',
                               (arg_0,))

    def GetCapabilities(self):
        return new_method_call(self, 'GetCapabilities')

    def GetServerInformation(self):
        return new_method_call(self, 'GetServerInformation')
# ---- End auto generated code ----


async def send_notification():
    async with open_dbus_router() as router:
        proxy = Proxy(Notifications(), router)

        resp = await proxy.Notify('jeepney_test',  # App name
                                  0,      # Not replacing any previous notification
                                  '',     # Icon
                                  'Hello, world!',  # Summary
                                  'This is an example notification from Jeepney',
```

(continues on next page)

(continued from previous page)

```
                                [], {},   # Actions, hints
                                -1,       # expire_timeout (-1 = default)
                               )
        print('Notification ID:', resp[0])

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

This is more code for the simple use case here, but in a larger application collecting the message definitions together like this could make it clearer.

1.3 Sending & receiving file descriptors

New in version 0.7.

D-Bus allows sending file descriptors - references to open files, sockets, etc. To use this, use the blocking, multithreading or Trio integration and enable it (`enable_fds=True`) when connecting to D-Bus. If you enable FD support but the message bus can't or won't support it, *FDNegotiationError* will be raised.

To send a file descriptor, pass any object with a `.fileno()` method, such as an open file or socket, or a suitable integer. The file descriptor must not be closed before the message is sent.

A received file descriptor will be returned as a *FileDescriptor* object to help avoid leaking FDs. This can easily be converted to a file object (*to_file()*), a socket (*to_socket()*) or a plain integer (*to_raw_fd()*).

```
# Send a file descriptor for a temp file (normally not visible in any folder)
with TemporaryFile() as tf:
    msg = new_method_call(server, 'write_data', 'h', (tf,))
    await router.send_and_get_reply(msg)

# Receive a file descriptor, use it as a writable file
msg = await conn.receive()
fd, = msg.body
with fd.to_file('w') as f:
    f.write(f'Timestamp: {datetime.now()}')
```

The snippets above are based on the Trio integration. See the examples directory in the Jeepney repository for complete, working examples.

CHAPTER 2

Making and parsing messages

The core of Jeepney is code to build, serialise and deserialise D-Bus messages.

2.1 Making messages

D-Bus has four message types. Three, *method call*, *method return* and *error*, are used in a request-reply pattern. The fourth, *signal*, is a broadcast message with no reply.

- *Method call* messages are most conveniently made with a message generator class, which can be *autogenerated*. One layer down from this is `new_method_call()`, which takes a `DBusAddress` object.
- *Method return* and *error* messages are made with `new_method_return()` and `new_error()`, passing the method call message which they are replying to.
- *signal* messages are made with `new_signal()`, which takes a `DBusAddress` representing the sender.

All of these return a `Message` object. `Message.serialise()` converts it to bytes, but none of these core methods ever send a message. See the *integration layer* for that.

2.1.1 Signatures

D-Bus is strongly typed, and every message has a *signature* describing the body data. These are strings using characters such as `i` for a signed 32-bit integer. See the DBus specification for the full list.

Jeepney does not try to guess or discover the signature when you build a message: your code must explicitly specify a signature for every message. However, Jeepney can help you write this code: see *Generating D-Bus wrappers*.

D-Bus types are converted to and from native Python objects as follows:

- All the D-Bus integer types are represented as Python `int`, including *BYTE* when it's not in an array.
- *BOOLEAN* is `bool`.
- *DOUBLE* is `float`.
- *STRING*, *OBJECT_PATH* and *SIGNATURE* are all `str`.

- *ARRAY* is `list`, except that an array of *BYTE* is a `bytes` object, and an array of *DICT_ENTRY* is a `dict`.
- *STRUCT* is `tuple`.
- *VARIANT* is a 2-tuple `(signature, data)`. E.g. to put a string into a variant field, you would pass the data `("s", "my string")`.
- *UNIX_FD* are converted from objects with a `.fileno()` method or plain integers, and converted to *`FileDescriptor`* objects. See *Sending & receiving file descriptors* for more details.

CHAPTER 3

Generating D-Bus wrappers

D-Bus includes a mechanism to introspect remote objects and discover the methods they define. Jeepney can use this to generate classes defining the messages to send. Use it like this:

```
python3 -m jeepney.bindgen --name org.freedesktop.Notifications \
        --path /org/freedesktop/Notifications
```

This command will produce the class in the example under *Message generators and proxies*.

You specify *name*—which D-Bus service you're talking to—and *path*—an object in that service. Jeepney will generate a wrapper for each interface that object has, except for some standard ones like the introspection interface itself.

You are welcome to edit the generated code, e.g. to add docstrings or give parameters meaningful names. Names like `arg_1` are created when introspection doesn't provide a name.

CHAPTER 4

API reference

4.1 Core API

4.1.1 Message constructors

jeepney.**new_method_call**(*remote_obj*, *method*, *signature=None*, *body=()*)
Construct a new method call message

This is a relatively low-level method. In many cases, this will be called from a *MessageGenerator* subclass which provides a more convenient API.

Parameters

- **remote_obj** (DBusAddress) – The object to call a method on
- **method** (*str*) – The name of the method to call
- **signature** (*str*) – The DBus signature of the body data
- **body** (*tuple*) – Body data (i.e. method parameters)

jeepney.**new_method_return**(*parent_msg*, *signature=None*, *body=()*)
Construct a new response message

Parameters

- **parent_msg** (Message) – The method call this is a reply to
- **signature** (*str*) – The DBus signature of the body data
- **body** (*tuple*) – Body data

jeepney.**new_error**(*parent_msg*, *error_name*, *signature=None*, *body=()*)
Construct a new error response message

Parameters

- **parent_msg** (Message) – The method call this is a reply to

- **error_name** (*str*) – The name of the error
- **signature** (*str*) – The DBus signature of the body data
- **body** (*tuple*) – Body data

jeepney.**new_signal**(*emitter*, *signal*, *signature=None*, *body=()*)
Construct a new signal message

Parameters

- **emitter** (DBusAddress) – The object sending the signal
- **signal** (*str*) – The name of the signal
- **signature** (*str*) – The DBus signature of the body data
- **body** (*tuple*) – Body data

class jeepney.**DBusAddress**(*object_path*, *bus_name=None*, *interface=None*)
This identifies the object and interface a message is for.

e.g. messages to display desktop notifications would have this address:

```
DBusAddress('/org/freedesktop/Notifications',
            bus_name='org.freedesktop.Notifications',
            interface='org.freedesktop.Notifications')
```

class jeepney.**MessageGenerator**(*object_path*, *bus_name*)
Subclass this to define the methods available on a DBus interface.

jeepney.bindgen can automatically create subclasses using introspection.

See also:

Generating D-Bus wrappers

4.1.2 Parsing

class jeepney.**Parser**
Parse DBus messages from a stream of incoming data.

add_data(*data: bytes*, *fds=()*)
Provide newly received data to the parser

get_next_message() → Optional[jeepney.low_level.Message]
Parse one message, if there is enough data.

Returns None if it doesn't have a complete message.

4.1.3 Message objects

class jeepney.**Message**(*header*, *body*)
Object representing a DBus message.

It's not normally necessary to construct this directly: use higher level functions and methods instead.

header
A *Header* object

body
A tuple of the data in this message. The number and types of the elements depend on the message's signature:

D-Bus type	D-Bus code	Python type
BYTE	`y`	int
BOOLEAN	`b`	bool
INT16	`n`	int
UINT16	`q`	int
INT32	`i`	int
UINT32	`u`	int
INT64	`x`	int
UINT64	`t`	int
DOUBLE	`d`	float
STRING	`s`	str
OBJECT_PATH	`o`	str
SIGNATURE	`g`	str
ARRAY	`a`	list
STRUCT	`()`	tuple
VARIANT	`v`	2-tuple `(signature, value)`
DICT_ENTRY	`{}`	dict (for array of dict entries)
UNIX_FD	`h`	See *Sending & receiving file descriptors*

serialise(*serial=None*, *fds=None*) → bytes
Convert this message to bytes.

Specifying *serial* overrides the `msg.header.serial` field, so a connection can use its own serial number without modifying the message.

If file-descriptor support is in use, *fds* should be a `array.array` object with type `'i'`. Any file descriptors in the message will be added to the array. If the message contains FDs, it can't be serialised without this array.

class jeepney.**Header**(*endianness*, *message_type*, *flags*, *protocol_version*, *body_length*, *serial*, *fields*)

endianness
Endianness object, affecting message serialisation.

message_type
MessageType object.

flags
MessageFlag object.

protocol_version
Currently always 1.

body_length
The length of the raw message body in bytes.

serial
Sender's serial number for this message. This is not necessarily set for outgoing messages - see *Message.serialise()*.

fields
Mapping of *HeaderFields* values to the relevant Python objects.

4.1.4 Exceptions

exception jeepney.**SizeLimitError**
Raised when trying to (de-)serialise data exceeding D-Bus' size limit.

This is currently only implemented for arrays, where the maximum size is 64 MiB.

exception jeepney.**DBusErrorResponse**(*msg*)
Raised by proxy method calls when the reply is an error message

name
The error name from the remote end.

body
Any data fields contained in the error message.

4.1.5 Enums & Flags

class jeepney.**Endianness**

little = 1

big = 2

class jeepney.**HeaderFields**

path = 1

interface = 2

member = 3

error_name = 4

reply_serial = 5

destination = 6

sender = 7

signature = 8

unix_fds = 9

class jeepney.**MessageFlag**

no_reply_expected = 1
On a method call message, indicates that a reply should not be sent.

no_auto_start = 2
D-Bus includes a mechanism to start a service on demand to handle messages. If this flag is set, it will avoid that, only handling the message if the target is already running.

allow_interactive_authorization = 4
Signals that the recipient may prompt the user for elevated privileges to handle the request. The D-Bus specification has more details.

class jeepney.**MessageType**

method_call = 1

method_return = 2

error = 3

signal = 4

4.1.6 Matching messages

class jeepney.**MatchRule**(*, *type=None*, *sender=None*, *interface=None*, *member=None*, *path=None*, *path_namespace=None*, *destination=None*, *eavesdrop=False*)

Construct a match rule to subscribe to DBus messages.

e.g.:

```
mr = MatchRule(
    interface='org.freedesktop.DBus',
    member='NameOwnerChanged',
    type='signal'
)
msg = message_bus.AddMatch(mr)
# Send this message to subscribe to the signal
```

MatchRule objects are used both for filtering messages internally, and for setting up subscriptions in the message bus.

add_arg_condition(*argno: int*, *value: str*, *kind='string'*)

Add a condition for a particular argument

argno: int, 0-63 kind: 'string', 'path', 'namespace'

matches(*msg: jeepney.low_level.Message*) → bool

Returns True if msg matches this rule

serialise() → str

Convert to a string to use in an AddMatch call to the message bus

4.2 Common messages

These classes are *message generators*. Wrap them in a *Proxy* class to actually send the messages as well.

class jeepney.**Properties**(*obj: Union[jeepney.wrappers.DBusAddress, jeepney.wrappers.MessageGenerator]*)

Build messages for accessing object properties

If a D-Bus object has multiple interfaces, each interface has its own set of properties.

This uses the standard DBus interface `org.freedesktop.DBus.Properties`

get(*name*)

Get the value of the property *name*

get_all()

Get all property values for this interface

set(*name*, *signature*, *value*)

Set the property *name* to *value* (with appropriate signature)

class jeepney.**Introspectable**(*object_path*, *bus_name*)

Introspect()
Request D-Bus introspection XML for a remote object

class jeepney.**DBus**(*object_path='/org/freedesktop/DBus'*, *bus_name='org.freedesktop.DBus'*)
Messages to talk to the message bus

There is a ready-made instance of this at `jeepney.message_bus`.

AddMatch(*rule*)
rule can be a str or a *MatchRule* instance

GetAdtAuditSessionData(*name*)

GetConnectionCredentials(*name*)

GetConnectionSELinuxSecurityContext(*name*)

GetConnectionUnixProcessID(*name*)

GetConnectionUnixUser(*name*)

GetId()

GetNameOwner(*name*)

Hello()

ListActivatableNames()

ListNames()

ListQueuedOwners(*name*)

NameHasOwner(*name*)

ReleaseName(*name*)

ReloadConfig()

RemoveMatch(*rule*)

RequestName(*name*, *flags=0*)

StartServiceByName(*name*)

UpdateActivationEnvironment(*env*)

interface = 'org.freedesktop.DBus'

class jeepney.**Monitoring**(*object_path='/org/freedesktop/DBus'*, *bus_name='org.freedesktop.DBus'*)

BecomeMonitor(*rules*)
Convert this connection to a monitor connection (advanced)

4.3 Authentication

Note: If you use any of Jeepney's I/O integrations, authentication is built in. You only need these functions if you're working outside that.

If you are setting up a socket for D-Bus, you will need to do SASL authentication before starting to send and receive D-Bus messages. This text based protocol is completely different to D-Bus itself.

Only a small fraction of SASL is implemented here, primarily what Jeepney's integration layer uses. If you're doing something different, you may need to implement other messages yourself.

jeepney.auth.**make_auth_external**() → bytes
Prepare an AUTH command line with the current effective user ID.

This is the preferred authentication method for typical D-Bus connections over a Unix domain socket.

jeepney.auth.**make_auth_anonymous**() → bytes
Format an AUTH command line for the ANONYMOUS mechanism

Jeepney's higher-level wrappers don't currently use this mechanism, but third-party code may choose to.

See <https://tools.ietf.org/html/rfc4505> for details.

jeepney.auth.**BEGIN**
Send this just before switching to the D-Bus protocol.

class jeepney.auth.**Authenticator**(*enable_fds=False*)
Process data for the SASL authentication conversation

If enable_fds is True, this includes negotiating support for passing file descriptors.

Changed in version 0.7: This class was renamed from `SASLParser` and substantially changed.

authenticated
Initially False, changes to True when authentication has succeeded.

error
`None`, or the raw bytes of an error message if authentication failed.

data_to_send() → Optional[bytes]
Get a line of data to send to the server

The data returned should be sent before waiting to receive data. Returns empty bytes if waiting for more data from the server, and None if authentication is finished (success or error).

Iterating over the Authenticator object will also yield these lines; *`feed()`* should be called with received data inside the loop.

feed(*data: bytes*)
Process received data

Raises AuthenticationError if the incoming data is not as expected for successful authentication. The connection should then be abandoned.

exception jeepney.auth.**AuthenticationError**(*data*, *msg='Authentication failed'*)
Raised when DBus authentication fails

exception jeepney.auth.**FDNegotiationError**(*data*)
Raised when file descriptor support is requested but not available

4.3.1 Typical flow

1. Send the data from *`Authenticator.data_to_send()`* (or `for req_data in authenticator`).
2. Receive data from the server, pass to *`Authenticator.feed()`*.
3. Repeat 1 & 2 until *`Authenticator.authenticated`* is True, or the for loop exits.
4. Send *`BEGIN`*.
5. Start sending & receiving D-Bus messages.

4.4 File descriptor support

class jeepney.**FileDescriptor**(*fd*)
A file descriptor received in a D-Bus message

This wrapper helps ensure that the file descriptor is closed exactly once. If you don't explicitly convert or close the FileDescriptor object, it will close its file descriptor when it goes out of scope, and emit a ResourceWarning.

to_file(*mode*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)
Convert to a Python file object:

```
with fd.to_file('w') as f:
    f.write('xyz')
```

The arguments are the same as for the builtin `open()` function.

The *FileDescriptor* can't be used after calling this. Closing the file object will also close the file descriptor.

Note: If the descriptor does not refer to a regular file, or it doesn't have the right access mode, you may get strange behaviour or errors while using it.

You can use `os.stat()` and the `stat` module to check the type of object the descriptor refers to, and `fcntl.fcntl()` to check the access mode, e.g.:

```
stat.S_ISREG(os.stat(fd.fileno()).st_mode)  # Regular file?

status_flags = fcntl.fcntl(fd, fcntl.F_GETFL)
(status_flags & os.O_ACCMODE) == os.O_RDONLY  # Read-only?
```

---

to_socket()
Convert to a socket object

This returns a standard library `socket.socket()` object:

```
with fd.to_socket() as sock:
    b = sock.sendall(b'xyz')
```

The wrapper object can't be used after calling this. Closing the socket object will also close the file descriptor.

to_raw_fd()
Convert to the low-level integer file descriptor:

```
raw_fd = fd.to_raw_fd()
os.write(raw_fd, b'xyz')
os.close(raw_fd)
```

The *FileDescriptor* can't be used after calling this. The caller is responsible for closing the file descriptor.

fileno()
Get the integer file descriptor

This does not change the state of the *FileDescriptor* object, unlike the `to_*` methods.

close()
Close the file descriptor

This can safely be called multiple times, but will raise RuntimeError if called after converting it with one of the `to_*` methods.

This object can also be used in a `with` block, to close it on leaving the block.

exception jeepney.**NoFDError**
Raised by *FileDescriptor* methods if it was already closed/converted

4.5 Blocking I/O

This is a good option for simple scripts, where you don't need to do anything else while waiting for a D-Bus reply. If you will use D-Bus for multiple threads, or you want a nicer way to wait for signals, see *Blocking I/O with threads*.

jeepney.io.blocking.**open_dbus_connection**(*bus='SESSION'*, *enable_fds=False*, *auth_timeout=1.0*) → jeepney.io.blocking.DBusConnection
Connect to a D-Bus message bus

Pass `enable_fds=True` to allow sending & receiving file descriptors. An error will be raised if the bus does not allow this. For simplicity, it's advisable to leave this disabled unless you need it.

D-Bus has an authentication step before sending or receiving messages. This takes < 1 ms in normal operation, but there is a timeout so that client code won't get stuck if the server doesn't reply. *auth_timeout* configures this timeout in seconds.

class jeepney.io.blocking.**DBusConnection**(*sock: socket.socket*, *enable_fds=False*)

send(*message: jeepney.low_level.Message*, *serial=None*)
Serialise and send a *Message* object

receive(*, *timeout=None*) → jeepney.low_level.Message
Return the next available message from the connection

If the data is ready, this will return immediately, even if timeout<=0. Otherwise, it will wait for up to timeout seconds, or indefinitely if timeout is None. If no message comes in time, it raises TimeoutError.

send_and_get_reply(*message*, *, *timeout=None*, *unwrap=None*)
Send a message, wait for the reply and return it

Filters are applied to other messages received before the reply - see `add_filter()`.

recv_messages(*, *timeout=None*)
Receive one message and apply filters

See *filter()*. Returns nothing.

filter(*rule*, *, *queue: Optional[collections.deque] = None*, *bufsize=1*)
Create a filter for incoming messages

Usage:

```
with conn.filter(rule) as matches:
    # matches is a deque containing matched messages
    matching_msg = conn.recv_until_filtered(matches)
```

Parameters

- **rule** (jeepney.MatchRule) – Catch messages matching this rule
- **queue** (*collections.deque*) – Matched messages will be added to this
- **bufsize** (*int*) – If no deque is passed in, create one with this size

recv_until_filtered(*queue*, *, *timeout=None*) → jeepney.low_level.Message

Process incoming messages until one is filtered into queue

Pops the message from queue and returns it, or raises TimeoutError if the optional timeout expires. Without a timeout, this is equivalent to:

```
while len(queue) == 0:
    conn.recv_messages()
return queue.popleft()
```

In the other I/O modules, there is no need for this, because messages are placed in queues by a separate task.

Parameters

- **queue** (*collections.deque*) – A deque connected by *filter()*
- **timeout** (*float*) – Maximum time to wait in seconds

close()

Close the connection

Using `with open_dbus_connection()` will also close the connection on exiting the block.

class jeepney.io.blocking.**Proxy**(*msggen*, *connection*, *, *timeout=None*)

A blocking proxy for calling D-Bus methods

You can call methods on the proxy object, such as `bus_proxy.Hello()` to make a method call over D-Bus and wait for a reply. It will either return a tuple of returned data, or raise *DBusErrorResponse*. The methods available are defined by the message generator you wrap.

You can set a time limit on a call by passing `_timeout=` in the method call, or set a default when creating the proxy. The `_timeout` argument is not passed to the message generator. All timeouts are in seconds, and `TimeoutErrror` is raised if it expires before a reply arrives.

Parameters

- **msggen** – A message generator object
- **connection** (DBusConnection) – Connection to send and receive messages
- **timeout** (*float*) – Default seconds to wait for a reply, or None for no limit

See also:

Message generators and proxies

4.6 Blocking I/O with threads

This allows using a D-Bus connection from multiple threads. The router also launches a separate thread to receive incoming messages. See *Connections and Routers* for more about the two interfaces.

jeepney.io.threading.**open_dbus_router**(*bus='SESSION'*, *enable_fds=False*)

Open a D-Bus 'router' to send and receive messages.

Use as a context manager:

```
with open_dbus_router() as router:
    ...
```

On leaving the `with` block, the connection will be closed.

Parameters

- **bus** (*str*) – 'SESSION' or 'SYSTEM' or a supported address.
- **enable_fds** (*bool*) – Whether to enable passing file descriptors.

Returns *DBusRouter*

class jeepney.io.threading.**DBusRouter**(*conn: jeepney.io.threading.DBusConnection*)

A client D-Bus connection which can wait for replies.

This runs a separate receiver thread and dispatches received messages.

It's possible to wrap a *DBusConnection* in a router temporarily. Using the connection directly while it is wrapped is not supported, but you can use it again after the router is closed.

send(*message*, *, *serial=None*)

Serialise and send a *Message* object

send_and_get_reply(*msg: jeepney.low_level.Message*, *, *timeout=None*) → jeepney.low_level.Message

Send a method call message, wait for and return a reply

filter(*rule*, *, *queue: Optional[queue.Queue] = None*, *bufsize=1*)

Create a filter for incoming messages

Usage:

```
with router.filter(rule) as queue:
    matching_msg = queue.get()
```

Parameters

- **rule** (jeepney.MatchRule) – Catch messages matching this rule
- **queue** (*queue.Queue*) – Matched messages will be added to this
- **bufsize** (*int*) – If no queue is passed in, create one with this size

close()

Close this router

This does not close the underlying connection.

Leaving the `with` block will also close the router.

class jeepney.io.threading.**Proxy**(*msggen*, *router*, *, *timeout=None*)

A blocking proxy for calling D-Bus methods via a *DBusRouter*.

You can call methods on the proxy object, such as `bus_proxy.Hello()` to make a method call over D-Bus and wait for a reply. It will either return a tuple of returned data, or raise *DBusErrorResponse*. The methods available are defined by the message generator you wrap.

You can set a time limit on a call by passing `_timeout=` in the method call, or set a default when creating the proxy. The `_timeout` argument is not passed to the message generator. All timeouts are in seconds, and `TimeoutErrror` is raised if it expires before a reply arrives.

Parameters

- **msggen** – A message generator object
- **router** (*DBusRouter*) – Router to send and receive messages
- **timeout** (*float*) – Default seconds to wait for a reply, or None for no limit

See also:

Message generators and proxies

jeepney.io.threading.**open_dbus_connection**(*bus='SESSION'*, *enable_fds=False*, *auth_timeout=1.0*)

Open a plain D-Bus connection

D-Bus has an authentication step before sending or receiving messages. This takes < 1 ms in normal operation, but there is a timeout so that client code won't get stuck if the server doesn't reply. *auth_timeout* configures this timeout in seconds.

Returns *DBusConnection*

class jeepney.io.threading.**DBusConnection**(*sock: socket.socket*, *enable_fds=False*)

send(*message: jeepney.low_level.Message*, *serial=None*)
Serialise and send a *Message* object

receive(*, *timeout=None*) → jeepney.low_level.Message
Return the next available message from the connection

If the data is ready, this will return immediately, even if timeout<=0. Otherwise, it will wait for up to timeout seconds, or indefinitely if timeout is None. If no message comes in time, it raises TimeoutError.

If the connection is closed from another thread, this will raise ReceiveStopped.

close()
Close the connection

4.7 Trio integration

This supports D-Bus in applications built with Trio. See *Connections and Routers* for more about the two interfaces.

jeepney.io.trio.**open_dbus_router**(*bus='SESSION'*, *, *enable_fds=False*)
Open a D-Bus 'router' to send and receive messages.

Use as an async context manager:

```
async with open_dbus_router() as req:
    ...
```

Parameters **bus** (*str*) – 'SESSION' or 'SYSTEM' or a supported address.

Returns *DBusRouter*

This is a shortcut for:

```
conn = await open_dbus_connection()
async with conn:
    async with conn.router() as req:
        ...
```

class jeepney.io.trio.**DBusRouter**(*conn: jeepney.io.trio.DBusConnection*)
A client D-Bus connection which can wait for replies.

This runs a separate receiver task and dispatches received messages.

send(*message*, *, *serial=None*)
Send a message, don't wait for a reply

send_and_get_reply(*message*) → jeepney.low_level.Message
Send a method call message and wait for the reply

Returns the reply message (method return or error message type).

filter(*rule*, *, *channel: Optional[trio.MemorySendChannel] = None*, *bufsize=1*)
Create a filter for incoming messages

Usage:

```
async with router.filter(rule) as receive_channel:
    matching_msg = await receive_channel.receive()

# OR:
send_chan, recv_chan = trio.open_memory_channel(1)
async with router.filter(rule, channel=send_chan):
    matching_msg = await recv_chan.receive()
```

If the channel fills up, The sending end of the channel is closed when leaving the `async with` block, whether or not it was passed in.

Parameters

- **rule** (*jeepney.MatchRule*) – Catch messages matching this rule
- **channel** (*trio.MemorySendChannel*) – Send matching messages here
- **bufsize** (*int*) – If no channel is passed in, create one with this size

aclose()
Stop the sender & receiver tasks

Leaving the `async with` block will also close the router.

class jeepney.io.trio.**Proxy**(*msggen*, *router*)
A trio proxy for calling D-Bus methods

You can call methods on the proxy object, such as `await bus_proxy.Hello()` to make a method call over D-Bus and wait for a reply. It will either return a tuple of returned data, or raise *DBusErrorResponse*. The methods available are defined by the message generator you wrap.

Parameters

- **msggen** – A message generator object.
- **router** (*DBusRouter*) – Router to send and receive messages.

See also:

Message generators and proxies

jeepney.io.trio.**open_dbus_connection**(*bus='SESSION'*, *, *enable_fds=False*) → jeepney.io.trio.DBusConnection
Open a plain D-Bus connection

Returns *DBusConnection*

class jeepney.io.trio.**DBusConnection**(*socket*, *enable_fds=False*)
A plain D-Bus connection with no matching of replies.

This doesn't run any separate tasks: sending and receiving are done in the task that calls those methods. It's suitable for implementing servers: several worker tasks can receive requests and send replies. For a typical client pattern, see *DBusRouter*.

Implements trio's channel interface for Message objects.

send(*message: jeepney.low_level.Message*, *, *serial=None*)
Serialise and send a *Message* object

receive() → jeepney.low_level.Message
Return the next available message from the connection

router()
Temporarily wrap this connection as a *DBusRouter*

To be used like:

```
async with conn.router() as req:
    reply = await req.send_and_get_reply(msg)
```

While the router is running, you shouldn't use *receive()*. Once the router is closed, you can use the plain connection again.

aclose()
Close the D-Bus connection

4.8 Asyncio integration

This supports D-Bus in applications built with asyncio. See *Connections and Routers* for more about the two interfaces.

jeepney.io.asyncio.**open_dbus_router**(*bus='SESSION'*)
Open a D-Bus 'router' to send and receive messages

Use as an async context manager:

```
async with open_dbus_router() as router:
    ...
```

class jeepney.io.asyncio.**DBusRouter**(*conn: jeepney.io.asyncio.DBusConnection*)
A 'client' D-Bus connection which can wait for a specific reply.

This runs a background receiver task, and makes it possible to send a request and wait for the relevant reply.

send(*message*, *, *serial=None*)
Send a message, don't wait for a reply

send_and_get_reply(*message*) → jeepney.low_level.Message
Send a method call message and wait for the reply

Returns the reply message (method return or error message type).

filter(*rule*, *, *queue: Optional[asyncio.queues.Queue] = None*, *bufsize=1*)
Create a filter for incoming messages

Usage:

```
with router.filter(rule) as queue:
    matching_msg = await queue.get()
```

Parameters

- **rule** (MatchRule) – Catch messages matching this rule
- **queue** (*asyncio.Queue*) – Send matching messages here
- **bufsize** (*int*) – If no queue is passed in, create one with this size

class jeepney.io.asyncio.**Proxy**(*msggen*, *router*)

An asyncio proxy for calling D-Bus methods

You can call methods on the proxy object, such as `await bus_proxy.Hello()` to make a method call over D-Bus and wait for a reply. It will either return a tuple of returned data, or raise *DBusErrorResponse*. The methods available are defined by the message generator you wrap.

Parameters

- **msggen** – A message generator object.
- **router** (DBusRouter) – Router to send and receive messages.

See also:

Message generators and proxies

jeepney.io.asyncio.**open_dbus_connection**(*bus='SESSION'*)

Open a plain D-Bus connection

Returns *DBusConnection*

class jeepney.io.asyncio.**DBusConnection**(*reader: asyncio.streams.StreamReader*, *writer: asyncio.streams.StreamWriter*)

A plain D-Bus connection with no matching of replies.

This doesn't run any separate tasks: sending and receiving are done in the task that calls those methods. It's suitable for implementing servers: several worker tasks can receive requests and send replies. For a typical client pattern, see *DBusRouter*.

send(*message: jeepney.low_level.Message*, *, *serial=None*)

Serialise and send a *Message* object

receive() → jeepney.low_level.Message

Return the next available message from the connection

close()

Close the D-Bus connection

4.9 I/O Exceptions

exception jeepney.io.**RouterClosed**

Raised in tasks waiting for a reply when the router is closed

This will also be raised if the receiver task crashes, so tasks are not stuck waiting for a reply that can never come. The router object will not be usable after this is raised.

There is also a deprecated `jeepney.io.tornado` integration. Recent versions of Tornado are built on asyncio, so you can use the asyncio integration with Tornado applications.

CHAPTER 5

Design & Limitations

There are two parts to Jeepney:

The **core** is all about creating D-Bus messages, serialising them to bytes, and deserialising bytes into `Message` objects. It aims to be a complete & reliable implementation of the D-Bus wire protocol. It follows the idea of "Sans-I/O", implementing the D-Bus protocol independent of any means of sending or receiving the data.

The second part is **I/O integration**. This supports the typical use case for D-Bus - connecting to a message bus on a Unix socket - with various I/O frameworks. There is one integration module for each framework, and they provide similar interfaces (*Connections and Routers*), but differ as much as necessary to fit in with the different frameworks - e.g. the Trio integration uses channels where the asyncio integration uses queues.

Jeepney also allows for a similar split in code using it. If you want to wrap the desktop notifications service, for instance, you can write (or generate) a *message generator* class for it. The same message generator class can then be wrapped in a *proxy* for any of Jeepney's I/O integrations.

5.1 Non-goals

Jeepney does not (currently) aim for:

- Very high performance. Parsing binary messages in pure Python code is not the fastest way to do it, but for many use cases of D-Bus it's more than fast enough.
- Supporting all possible D-Bus transports. The I/O integration layer only works with Unix sockets, the most common way to use D-Bus. If you need to use another transport, you can still use `Message.serialise()` and `Parser`, and deal with sending & receiving data yourself.
- Supporting all authentication options. The *auth module* only provides what the I/O integration layer uses.
- High-level server APIs. Jeepney's API for D-Bus servers is on a low-level, sending and receiving messages, not registering handler methods. See dbus-objects for a server API built on top of Jeepney.
- 'Magic' introspection. Some D-Bus libraries use introspection at runtime to discover available methods, but Jeepney does not. Instead, it uses introspection during development to write message generators (*Generating D-Bus wrappers*).

5.2 Alternatives

- GTK applications can use Gio.DBusConnection or a higher-level wrapper like dasbus or pydbus. There are also GObject wrappers for specific D-Bus services, e.g. secret storage and desktop notifications.
- PyQt applications can use the Qt D-Bus module. This has been available in PyQt for many years, and in PySide from version 6.2 (released in 2021).
- DBussy works with asyncio. It is a Python binding to the libdbus reference implementation in C, whereas Jeepney reimplements the D-Bus protocol in Python.
- dbus-python is the original Python binding to libdbus. It is very complete and well tested, but may be trickier to install and to integrate with event loops and async frameworks.

See also:

D-Bus Python bindings on the Freedesktop wiki

CHAPTER 6

What is D-Bus?

D-Bus is a system for programs on the same computer to communicate. It's used primarily on Linux, to interact with various parts of the operating system.

For example, take desktop notifications - the alerts that appear to tell you about things like new chat messages.

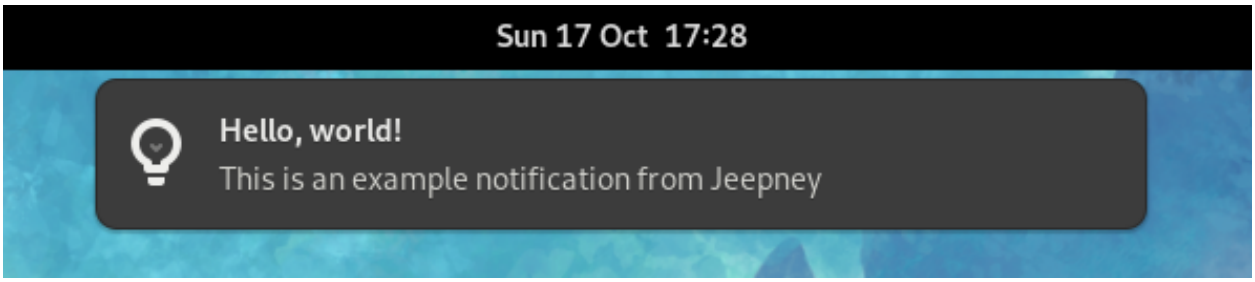

Fig. 1: A desktop notification on GNOME

A program that wants to display a notification sends a D-Bus message to the 'notification server', which displays an alert for a brief time and then hides it again. Different desktops, like GNOME and KDE, have different notification servers, but they handle the same messages (defined in the desktop notification spec), so programs don't need to do different things for different desktops.

Other things that use D-Bus include:

- Retrieving passwords from the desktop's 'keyring'
- Disabling the screensaver while playing a film
- Special keyboard keys, like pause & skip track, working with whichever media player you use.
- Opening a user's files in a sandboxed (Flatpak) application.

6.1 Methods & signals

D-Bus uses two types of messaging:

Method calls go to a specific destination, which replies with either a 'method return' or an error message. In the notifications example above, the program sends a method call message to the notification server to ask it to display a notification.

Signals are sent out to any program that subscribes to them. For example, when a desktop notification is closed, the notification server sends a signal. The application might use this to choose between updating the notification ('**2** new messages') or sending a new one. There's no reply to a signal, and the sender doesn't know if anything received it or not.

6.2 Names

There are a lot of names in D-Bus, and they can look quite similar. For instance, displaying a desktop notification involves sending a message to the bus name `org.freedesktop.Notifications`, for the object `/org/freedesktop/Notifications`, with the interface `org.freedesktop.Notifications`. What do those all mean?

- The bus name (`.` separated) is which program you're talking to.
- The object name (`/` separated) is which thing inside that program you want to use, e.g. which password in the keyring.
- The interface name (`.` separated) is which set of methods and signals you are using. Most objects have one main interface plus a few standard ones for things like introspection (finding what methods are available).

Finally, a simple name like `Notify` or `NotificationClosed` identifies which method is being called, or which signal is being sent, from a list for that interface.

The bus, object and interface names are all based on reversed domain names. The people who control https://freedesktop.org/ can define names starting with `org.freedesktop.` (or `/org/freedesktop/` for objects). There's no way to enforce this, but so long as everyone sticks to it, we don't have to worry about the same name being used for different things.

6.3 Message buses

Applications using D-Bus connect to a *message bus*, a small program which is always running. The bus takes care of delivering messages to other applications.

There are normally two buses you need to know about. Each logged-in user has their own **session bus**, handling things like desktop notifications (and the other examples above).

The **system bus** is shared for all users. In particular, requests sent via the system bus can do things that would otherwise require admin (sudo) access, like unmounting a USB stick or installing new packages. (How the system decides whether to allow these actions or not is a separate topic - look up 'polkit' if you want to know about that).

You can also talk to the message bus itself (using D-Bus messages, of course). This is how you subscribe to signals, or claim a bus name so other programs can send you method calls. The message bus has the name `org.freedesktop.DBus`.

Note: Programs *can* agree some other way to connect and send each other D-Bus messages without a message bus. This isn't very common, though.

6.4 Special features

You can send a D-Bus message to a program that's not even running, and the message bus will start it and then deliver the message. This feature (*activation*) means that programs don't have to stay running just to reply to D-Bus method calls. A config file installed with the application defines its bus name and how to launch it.

Because D-Bus is designed to be used between programs on the same computer, it can do things that are impossible over the network. D-Bus messages can include 'file descriptors', handles for things like open files, pipes and sockets. This can be used to selectively give a program access to something that would normally be off limits. See *Sending & receiving file descriptors* for how to use this from Jeepney.

See also:

Introduction to D-Bus (freedesktop.org)

Introduction to D-Bus (KDE)

D-Bus overview (txdbus)

CHAPTER 7

Release notes

7.1 0.8

2022-04-03

- Removed `jeepney.integrate` APIs, which were deprecated in 0.7. Use `jeepney.io` instead (see *Connecting to DBus and sending messages*).
- Removed deprecated `jeepney.io.tornado` API. Tornado now uses the asyncio event loop, so you can use it along with `jeepney.io.asyncio`.
- Deprecated `conn.router` attribute in the *Blocking I/O* integration. Use *proxies* or `send_and_get_reply()` to find replies to method calls, and `filter()` for other routing.
- Added docs page with background on D-Bus (*What is D-Bus?*).

7.2 0.7.1

2021-07-28

- Add `async with` support to `DBusConnection` in the asyncio integration.
- Fix calling `receive()` immediately after opening a connection in the asyncio integration.

Thanks to Aleksandr Mezin for these changes.

7.3 0.7

2021-07-21

- Support for *sending and receiving file descriptors*. This is available with the blocking, threading and trio integration layers.

- Deprecated older integration APIs, in favour of new APIs introduced in 0.5.
- Fixed passing a deque in to `filter()` in the blocking integration API.

7.4 0.6

2020-11-19

- New method `recv_until_filtered()` in the blocking I/O integration to receive messages until one is filtered into a queue.
- More efficient buffering of received data waiting to be parsed into D-Bus messages.

7.5 0.5

2020-11-10

- New common scheme for I/O integration - see *Connections and Routers*.
  - This is designed for tasks to wait for messages and then act on them, rather than triggering callbacks. This is based on ideas from 'structured concurrency', which also informs the design of Trio. See this blog post by Nathaniel Smith for more background.
  - There are new integrations for *Trio* and *threading*.
  - The old integration interfaces should still work for now, but they will be deprecated and eventually removed.
- `Message.serialise()` accepts a serial number, to serialise outgoing messages without modifying the message object.
- Improved documentation, including *API docs*.

7.6 0.4.3

2020-03-04

- The blocking integration now throws `ConnectionResetError` on all systems when the connection was closed from the other end. It would previously hang on some systems.

7.7 0.4.2

2020-01-03

- The blocking `DBusConnection` integration class now has a `.close()` method, and can be used as a context manager:

```
from jeepney.integrate.blocking import connect_and_authenticate
with connect_and_authenticate() as connection:
    ...
```

7.8 0.4.1

2019-08-11

- Avoid using `asyncio.Future` for the blocking integration.
- Set the 'destination' field on method return and error messages to the 'sender' from the parent message.

Thanks to Oscar Caballero and Thomas Grainger for contributing to this release.

7.9 0.4

2018-09-24

- Authentication failures now raise a new `AuthenticationError` subclass of `ValueError`, so that they can be caught specifically.
- Fixed logic error when authentication is rejected.
- Use *effective* user ID for authentication instead of *real* user ID. In typical use cases these are the same, but where they differ, effective uid seems to be the relevant one.
- The 64 MiB size limit for an array is now checked when serialising it.
- New function *`jeepney.auth.make_auth_anonymous()`* to prepare an anonymous authentication message. This is not used by the wrappers in Jeepney at the moment, but may be useful for third party code in some situations.
- New examples for subscribing to D-Bus signals, with blocking I/O and with asyncio.
- Various improvements to documentation.

Thanks to Jane Soko and Gitlab user xiretza for contributing to this release.

See also:

D-Feet App for exploring available D-Bus services on your machine.

D-Bus Specification Technical details about the D-Bus protocol.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

jeepney, 11
jeepney.auth, 17
jeepney.io.asyncio, 24
jeepney.io.blocking, 19
jeepney.io.threading, 20
jeepney.io.trio, 22

Index

A

aclose() (*jeepney.io.trio.DBusConnection method*), 24
aclose() (*jeepney.io.trio.DBusRouter method*), 23
add_arg_condition() (*jeepney.MatchRule method*), 15
add_data() (*jeepney.Parser method*), 12
AddMatch() (*jeepney.DBus method*), 16
allow_interactive_authorization (*jeepney.MessageFlag attribute*), 14
authenticated (*jeepney.auth.Authenticator attribute*), 17
AuthenticationError, 17
Authenticator (*class in jeepney.auth*), 17

B

BecomeMonitor() (*jeepney.Monitoring method*), 16
BEGIN (*in module jeepney.auth*), 17
big (*jeepney.Endianness attribute*), 14
body (*jeepney.DBusErrorResponse attribute*), 14
body (*jeepney.Message attribute*), 12
body_length (*jeepney.Header attribute*), 13

C

close() (*jeepney.FileDescriptor method*), 18
close() (*jeepney.io.asyncio.DBusConnection method*), 25
close() (*jeepney.io.blocking.DBusConnection method*), 20
close() (*jeepney.io.threading.DBusConnection method*), 22
close() (*jeepney.io.threading.DBusRouter method*), 21

D

data_to_send() (*jeepney.auth.Authenticator method*), 17
DBus (*class in jeepney*), 16
DBusAddress (*class in jeepney*), 12
DBusConnection (*class in jeepney.io.asyncio*), 25
DBusConnection (*class in jeepney.io.blocking*), 19
DBusConnection (*class in jeepney.io.threading*), 22
DBusConnection (*class in jeepney.io.trio*), 23
DBusErrorResponse, 14
DBusRouter (*class in jeepney.io.asyncio*), 24
DBusRouter (*class in jeepney.io.threading*), 21
DBusRouter (*class in jeepney.io.trio*), 22
destination (*jeepney.HeaderFields attribute*), 14

E

Endianness (*class in jeepney*), 14
endianness (*jeepney.Header attribute*), 13
error (*jeepney.auth.Authenticator attribute*), 17
error (*jeepney.MessageType attribute*), 15
error_name (*jeepney.HeaderFields attribute*), 14

F

FDNegotiationError, 17
feed() (*jeepney.auth.Authenticator method*), 17
fields (*jeepney.Header attribute*), 13
FileDescriptor (*class in jeepney*), 18
fileno() (*jeepney.FileDescriptor method*), 18
filter() (*jeepney.io.asyncio.DBusRouter method*), 24
filter() (*jeepney.io.blocking.DBusConnection method*), 19
filter() (*jeepney.io.threading.DBusRouter method*), 21
filter() (*jeepney.io.trio.DBusRouter method*), 23
flags (*jeepney.Header attribute*), 13

G

get() (*jeepney.Properties method*), 15
get_all() (*jeepney.Properties method*), 15
get_next_message() (*jeepney.Parser method*), 12
GetAdtAuditSessionData() (*jeepney.DBus method*), 16
GetConnectionCredentials() (*jeepney.DBus method*), 16
GetConnectionSELinuxSecurityContext() (*jeepney.DBus method*), 16

GetConnectionUnixProcessID() (*jeepney.DBus method*), 16
GetConnectionUnixUser() (*jeepney.DBus method*), 16
GetId() (*jeepney.DBus method*), 16
GetNameOwner() (*jeepney.DBus method*), 16

H

Header (*class in jeepney*), 13
header (*jeepney.Message attribute*), 12
HeaderFields (*class in jeepney*), 14
Hello() (*jeepney.DBus method*), 16

I

interface (*jeepney.DBus attribute*), 16
interface (*jeepney.HeaderFields attribute*), 14
Introspect() (*jeepney.Introspectable method*), 15
Introspectable (*class in jeepney*), 15

J

jeepney (*module*), 11
jeepney.auth (*module*), 17
jeepney.io.asyncio (*module*), 24
jeepney.io.blocking (*module*), 19
jeepney.io.threading (*module*), 20
jeepney.io.trio (*module*), 22

L

ListActivatableNames() (*jeepney.DBus method*), 16
ListNames() (*jeepney.DBus method*), 16
ListQueuedOwners() (*jeepney.DBus method*), 16
little (*jeepney.Endianness attribute*), 14

M

make_auth_anonymous() (*in module jeepney.auth*), 17
make_auth_external() (*in module jeepney.auth*), 17
matches() (*jeepney.MatchRule method*), 15
MatchRule (*class in jeepney*), 15
member (*jeepney.HeaderFields attribute*), 14
Message (*class in jeepney*), 12
message_type (*jeepney.Header attribute*), 13
MessageFlag (*class in jeepney*), 14
MessageGenerator (*class in jeepney*), 12
MessageType (*class in jeepney*), 14
method_call (*jeepney.MessageType attribute*), 14
method_return (*jeepney.MessageType attribute*), 14
Monitoring (*class in jeepney*), 16

N

name (*jeepney.DBusErrorResponse attribute*), 14
NameHasOwner() (*jeepney.DBus method*), 16
new_error() (*in module jeepney*), 11
new_method_call() (*in module jeepney*), 11
new_method_return() (*in module jeepney*), 11
new_signal() (*in module jeepney*), 12
no_auto_start (*jeepney.MessageFlag attribute*), 14
no_reply_expected (*jeepney.MessageFlag attribute*), 14
NoFDError, 19

O

open_dbus_connection() (*in module jeepney.io.asyncio*), 25
open_dbus_connection() (*in module jeepney.io.blocking*), 19
open_dbus_connection() (*in module jeepney.io.threading*), 22
open_dbus_connection() (*in module jeepney.io.trio*), 23
open_dbus_router() (*in module jeepney.io.asyncio*), 24
open_dbus_router() (*in module jeepney.io.threading*), 20
open_dbus_router() (*in module jeepney.io.trio*), 22

P

Parser (*class in jeepney*), 12
path (*jeepney.HeaderFields attribute*), 14
Properties (*class in jeepney*), 15
protocol_version (*jeepney.Header attribute*), 13
Proxy (*class in jeepney.io.asyncio*), 25
Proxy (*class in jeepney.io.blocking*), 20
Proxy (*class in jeepney.io.threading*), 21
Proxy (*class in jeepney.io.trio*), 23

R

receive() (*jeepney.io.asyncio.DBusConnection method*), 25
receive() (*jeepney.io.blocking.DBusConnection method*), 19
receive() (*jeepney.io.threading.DBusConnection method*), 22
receive() (*jeepney.io.trio.DBusConnection method*), 24
recv_messages() (*jeepney.io.blocking.DBusConnection method*), 19
recv_until_filtered() (*jeepney.io.blocking.DBusConnection method*), 20
ReleaseName() (*jeepney.DBus method*), 16
ReloadConfig() (*jeepney.DBus method*), 16
RemoveMatch() (*jeepney.DBus method*), 16
reply_serial (*jeepney.HeaderFields attribute*), 14

RequestName() (*jeepney.DBus method*), 16
router() (*jeepney.io.trio.DBusConnection method*), 24
RouterClosed, 25

S

send() (*jeepney.io.asyncio.DBusConnection method*), 25
send() (*jeepney.io.asyncio.DBusRouter method*), 24
send() (*jeepney.io.blocking.DBusConnection method*), 19
send() (*jeepney.io.threading.DBusConnection method*), 22
send() (*jeepney.io.threading.DBusRouter method*), 21
send() (*jeepney.io.trio.DBusConnection method*), 24
send() (*jeepney.io.trio.DBusRouter method*), 23
send_and_get_reply() (*jeepney.io.asyncio.DBusRouter method*), 24
send_and_get_reply() (*jeepney.io.blocking.DBusConnection method*), 19
send_and_get_reply() (*jeepney.io.threading.DBusRouter method*), 21
send_and_get_reply() (*jeepney.io.trio.DBusRouter method*), 23
sender (*jeepney.HeaderFields attribute*), 14
serial (*jeepney.Header attribute*), 13
serialise() (*jeepney.MatchRule method*), 15
serialise() (*jeepney.Message method*), 13
set() (*jeepney.Properties method*), 15
signal (*jeepney.MessageType attribute*), 15
signature (*jeepney.HeaderFields attribute*), 14
SizeLimitError, 14
StartServiceByName() (*jeepney.DBus method*), 16

T

to_file() (*jeepney.FileDescriptor method*), 18
to_raw_fd() (*jeepney.FileDescriptor method*), 18
to_socket() (*jeepney.FileDescriptor method*), 18

U

unix_fds (*jeepney.HeaderFields attribute*), 14
UpdateActivationEnvironment() (*jeepney.DBus method*), 16