Jeepney Documentation

Release 0.4.1

Thomas Kluyver

Aug 11, 2019

Contents

Jeepney is a pure Python interface to D-Bus, a protocol for interprocess communication on desktop Linux (mostly).

The core of Jeepney is I/O free, and the `jeepney.integrate` 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`. To use it, the DBus 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

So far, Jeepney can be used with three different I/O systems:

- Blocking (synchronous) I/O
- asyncio
- Tornado

For each of these, there is a module in `jeepney.integrate` which exposes a function called `connect_and_authenticate`. This establishes a DBus connection and returns an object you can use to send and receive messages. Exactly what it returns may vary, though.

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

```
from jeepney import DBusAddress, new_method_call
from jeepney.integrate.blocking import connect_and_authenticate

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

connection = connect_and_authenticate(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[0])
```

And here is the same thing using asyncio:

```
import asyncio

from jeepney import DBusAddress, new_method_call
from jeepney.integrate.asyncio import connect_and_authenticate

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

async def send_notification():
    (transport, protocol) = await connect_and_authenticate(bus='SESSION')

    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
    reply = await protocol.send_message(msg)
    print('Notification ID:', reply[0])

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

1.1 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:

```
import asyncio

from jeepney import MessageGenerator, new_method_call
from jeepney.integrate.asyncio import connect_and_authenticate, 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)
```

(continues on next page)

(continued from previous page)

```
    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():
    (transport, protocol) = await connect_and_authenticate(bus='SESSION')
    proxy = Proxy(Notifications(), protocol)

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

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(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.

CHAPTER 2

Limitations

Some lesser-used parts of the D-Bus spec are not implemented:

1. Jeepney only connects to Unix domain sockets. This is how D-Bus is normally exposed, but the specification allows for other transports, such as TCP sockets, which Jeepney does not support.
2. Only the 'external' auth method is used. The specification recommends this mechanism where it's available, and it's the obvious thing to use with Unix domain sockets.
3. Sending and receiving Unix file descriptors is not supported.

Any of these limitations may be lifted in the future, if there's a need and we can find a clean way to do so. If you want to remove a limitation, be prepared to get involved. :-)

CHAPTER 3

Making and parsing messages

The core of Jeepney is code to build, serialise and deserialise DBus messages.

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.

serialise()
Convert this message to bytes.

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

feed(*data*)
Feed the parser newly read data.

Returns a list of messages completed by the new data.

3.1 Making messages

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')
```

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

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
- **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

See also:

Message generators and proxies

3.1.1 Signatures

DBus 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*.

In most cases, DBus types have an obvious corresponding type in Python. However, a few types require further explanation:

- DBus *ARRAY* are Python lists, except for arrays of *DICT_ENTRY*, which are dicts.
- DBus *STRUCT* are Python tuples.
- DBus *VARIANT* are 2-tuples `(signature, data)`. E.g. to put a string into a variant field, you would pass the data `("s", "my string")`.
- Jeepney does not (yet) support sending or receiving file descriptors.

CHAPTER 4

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 code used in the previous page (see *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.

CHAPTER 5

Release notes

5.1 0.4.1

- 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.

5.2 0.4

- 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 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Index