dasbus

Vendula Poncova

Jul 10, 2023

CONTENTS:

Dasbus is a DBus library written in Python 3, based on GLib and inspired by pydbus. The code used to be part of the Anaconda Installer project. It was based on the pydbus library, but we replaced it with our own solution because its upstream development stalled. The dasbus library is a result of this effort.

CHAPTER

ONE

REQUIREMENTS

- Python 3.6+
- PyGObject 3

You can install PyGObject provided by your operating system or use PyPI. The system package is usually called `python3-gi`, `python3-gobject` or `pygobject3`. See the instructions for your platform (only for PyGObject, you don't need cairo or GTK).

The library is known to work with Python 3.8, PyGObject 3.34 and GLib 2.63, but these are not the required minimal versions.

CHAPTER

TWO

INSTALLATION

Install the package from PyPI or install the package provided by your operating system if available.

2.1 Install from PyPI

Follow the instructions above to install the requirements before you install `dasbus` with `pip`. The required dependencies has to be installed manually in this case.

```
pip3 install dasbus
```

2.2 Install on Arch Linux

Build and install the community package from the Arch User Repository. Follow the guidelines.

```
git clone https://aur.archlinux.org/python-dasbus.git
cd python-dasbus
makepkg -si
```

2.3 Install on Debian / Ubuntu

Install the system package on Debian 11+ or Ubuntu 22.04+.

```
sudo apt install python3-dasbus
```

2.4 Install on Fedora / CentOS / RHEL

Install the system package on Fedora 31+, CentOS Stream 8+ or RHEL 8+.

```
sudo dnf install python3-dasbus
```

2.5 Install on openSUSE

Install the system package on openSUSE Tumbleweed or openSUSE Leap 15.2+.

```
sudo zypper install python3-dasbus
```

2.5.1 Development and testing

Install podman and use the following command to run all tests in a container. It doesn't require any additional dependencies:

```
make container-ci
```

Use the command below to run only the unit tests:

```
make container-ci CI_CMD="make test"
```

2.5.2 dasbus vs pydbus

The dasbus library used to be based on pydbus, but it was later reimplemented. We have changed the API and the implementation of the library based on our experience with pydbus. However, it should be possible to modify dasbus classes to work the same way as pydbus classes.

What is new

- Support for asynchronous DBus calls: DBus methods can be called asynchronously.
- Support for Unix file descriptors: It is possible to send or receive Unix file descriptors.
- Mapping DBus errors to exceptions: Use Python exceptions to propagate and handle DBus errors. Define your own rules for mapping errors to exceptions and back. See the *ErrorMapper* class
- Support for type hints: Use Python type hints from *dasbus.typing* to define DBus types.
- Generating XML specifications: Automatically generate XML specifications from Python classes with the *dbus_interface* decorator.
- Support for DBus structures: Represent DBus structures (dictionaries of variants) by Python objects. See the *DBusData* class.
- Support for groups of DBus objects: Use DBus containers from *dasbus.server.container* to publish groups of Python objects.
- Composition over inheritance: The library follows the principle of composition over inheritance. It allows to easily change the default behaviour.
- Lazy DBus connections: DBus connections are established on demand.
- Lazy DBus proxies: Attributes of DBus proxies are created on demand.

What is different

- No context managers: There are no context managers in dasbus. Context managers and event loops don't work very well together.
- No auto-completion: There is no support for automatic completion of DBus names and paths. We recommend to work with constants defined by classes from *dasbus.identifier* instead of strings.
- No unpacking of variants: The dasbus library doesn't unpack variants by default. It means that values received from DBus match the types declared in the XML specification. Use the *get_native* function to unpack the values.
- Obtaining proxy objects: Call the *get_proxy* method to get a proxy of the specified DBus object.
- No single-interface view: DBus proxies don't support single-interface views. Use the *InterfaceProxy* class to access a specific interface of a DBus object.
- Higher priority of standard interfaces: If there is a DBus interface in the XML specification that redefines a member of a standard interface, the DBus proxy will choose a member of the standard interface. Use the *InterfaceProxy* class to access a specific interface of a DBus object.
- No support for help: Members of DBus proxies are created lazily, so the build-in `help` function doesn't return useful information about the DBus interfaces.
- Watching DBus names: Use *a service observer* to watch a DBus name.
- Acquiring DBus names: Call the *register_service* method to acquire a DBus name.
- Providing XML specifications: Use the `__dbus_xml__` attribute to provide the XML specification of a DBus object. Or you can generate it from the code using the *dbus_interface* decorator.
- No support for polkit: There is no support for the DBus service `org.freedesktop.PolicyKit1`.

What is the same (for now)

- No support for other event loops: Dasbus uses GLib as its backend, so it requires to use the GLib event loop. However, the GLib part of dasbus is separated from the rest of the code, so it shouldn't be too difficult to add support for a different backend. It would be necessary to replace *dasbus.typing.Variant* and *dasbus.typing.VariantType* with their abstractions and reorganize the code.
- No support for org.freedesktop.DBus.ObjectManager: There is no support for object managers, however the *DBus containers* could be a good starting point.

2.5.3 dasbus package

Subpackages

dasbus.client package

Submodules

dasbus.client.handler module

class AbstractClientObjectHandler(*message_bus*, *service_name*, *object_path*)

Bases: `object`

The abstract handler of a remote DBus object.

create_member(*interface_name*, *member_name*)

Create a member of the DBus object.

Parameters

- **interface_name** – a name of the interface
- **member_name** – a name of the member

Returns
a signal, a method or a property

abstract disconnect_members()

Disconnect members of the DBus object.

Unsubscribe from DBus signals and disconnect all registered callbacks of the proxy signals.

property object_path

DBus object path.

Returns
a DBus path

property service_name

DBus service name.

Returns
a DBus name

property specification

DBus specification.

class ClientObjectHandler(*message_bus*, *service_name*, *object_path*, *error_mapper=None*, *client=<class 'dasbus.client.handler.GLibClient'>*, *signal_factory=<class 'dasbus.signal.Signal'>*)

Bases: *AbstractClientObjectHandler*

The client handler of a DBus object.

disconnect_members()

Disconnect members of the DBus object.

class GLibClient

Bases: object

The low-level DBus client library based on GLib.

DBUS_TIMEOUT_NONE = 2147483647

classmethod async_call(*connection*, *service_name*, *object_path*, *interface_name*, *method_name*, *parameters*, *reply_type*, *callback*, *callback_args=()*, *flags=0*, *timeout=2147483647*)

Asynchronously call a DBus method.

classmethod get_remote_error_message(*error*)

Get a message of the remote DBus error.

classmethod get_remote_error_name(*error*)

Get a DBus name of the remote DBus error.

classmethod is_remote_error(*error*)

Is it a remote DBus error?

classmethod is_timeout_error(*error*)

Is it a timeout error?

classmethod subscribe_signal(*connection*, *service_name*, *object_path*, *interface_name*, *signal_name*, *callback*, *callback_args=()*, *flags=0*)

Subscribe to a signal.

Returns
a callback to unsubscribe

classmethod sync_call(*connection*, *service_name*, *object_path*, *interface_name*, *method_name*, *parameters*, *reply_type*, *flags=0*, *timeout=2147483647*)

Synchronously call a DBus method.

Returns
a result of the DBus call

dasbus.client.observer module

class DBusObserver(*message_bus*, *service_name*, *monitoring=<class 'dasbus.client.observer.GLibMonitoring'>*)

Bases: `object`

Base class for DBus observers.

This class is recommended to use only to watch the availability of a service on DBus. It doesn't provide any support for accessing objects provided by the service.

Usage:

```
# Create the observer and connect to its signals.
observer = DBusObserver(SystemBus, "org.freedesktop.NetworkManager")

def callback1(observer):
    print("Service is available!")

def callback2(observer):
    print("Service is unavailable!")

observer.service_available.connect(callback1)
observer.service_unavailable.connect(callback2)

# Connect to the service once it is available.
observer.connect_once_available()

# Disconnect the observer.
observer.disconnect()
```

connect_once_available()

Connect to the service once it is available.

The observer is not connected to the service until it emits the service_available signal.

disconnect()

Disconnect from the service.

Disconnect from the service if it is connected and stop watching its availability.

property is_service_available

The proxy can be accessed.

property service_available

Signal that emits when the service is available.

Signal emits this class as an argument. You have to call the watch method to activate the signals.

property service_name

Returns a DBus name.

property service_unavailable

Signal that emits when the service is unavailable.

Signal emits this class as an argument. You have to call the watch method to activate the signals.

exception DBusObserverError

Bases: Exception

Exception class for the DBus observers.

class GLibMonitoring

Bases: object

The low-level DBus monitoring library based on GLib.

classmethod watch_name(*connection*, *name*, *flags=0*, *name_appeared=None*, *name_vanished=None*)

Watch a service name on the DBus connection.

dasbus.client.property module

class PropertyProxy(*getter*, *setter*)

Bases: object

Proxy of a remote DBus property.

It can be used to define instance attributes.

get()

Get the value of the DBus property.

set(*value*)

Set the value of the DBus property.

dasbus.client.proxy module

class AbstractObjectProxy(*message_bus*, *service_name*, *object_path*, *handler_factory=<class 'dasbus.client.handler.ClientObjectHandler'>*, ***handler_arguments*)

Bases: `object`

Abstract proxy of a remote DBus object.

class InterfaceProxy(*message_bus*, *service_name*, *object_path*, *interface_name*, **args*, ***kwargs*)

Bases: *AbstractObjectProxy*

Proxy of a remote DBus interface.

class ObjectProxy(**args*, ***kwargs*)

Bases: *AbstractObjectProxy*

Proxy of a remote DBus object.

disconnect_proxy(*proxy*)

Disconnect the DBus proxy from the remote object.

Parameters
proxy – a DBus proxy

get_object_path(*proxy*)

Get an object path of the remote DBus object.

Parameters
proxy – a DBus proxy

Returns
a DBus path

Module contents

dasbus.server package

Submodules

dasbus.server.container module

class DBusContainer(*message_bus*, *namespace*, *basename=None*)

Bases: `object`

The container of DBus objects.

A DBus container should be used to dynamically publish Publishable objects within the same namespace. It generates a unique DBus path for each object. It is able to resolve a DBus path into an object and an object into a DBus path.

Example:

```
# Create a container of tasks.
container = DBusContainer(
    namespace=("my", "project"),
    basename="Task",
```

(continues on next page)

(continued from previous page)

```
    message_bus=DBus
)

# Publish a task.
path = container.to_object_path(MyTask())

# Resolve an object path into a task.
task = container.from_object_path(path)
```

from_object_path(*object_path: ObjPath*)

Convert a DBus path to a published object.

If no published object is found for the given DBus path, raise DBusContainerError.

Parameters
object_path – a DBus path

Returns
a published object

from_object_path_list(*object_paths: List[ObjPath]*)

Convert DBus paths to published objects.

Parameters
object_paths – a list of DBus paths

Returns
a list of published objects

set_namespace(*namespace*)

Set the namespace.

All DBus objects from the container should use the same namespace, so the namespace should be set up before any of the DBus objects are published.

Parameters
namespace – a sequence of names

to_object_path(*obj*) → ObjPath

Convert a publishable object to a DBus path.

If no DBus path is found for the given object, publish the object on the container message bus with a unique DBus path generated from the container namespace.

Parameters
obj – a publishable object

Returns
a DBus path

to_object_path_list(*objects*) → List[ObjPath]

Convert publishable objects to DBus paths.

Parameters
objects – a list of publishable objects

Returns
a list of DBus paths

exception DBusContainerError

Bases: Exception

General exception for DBus container errors.

dasbus.server.handler module

class AbstractServerObjectHandler(*message_bus*, *object_path*, *obj*)

Bases: object

The abstract handler of a published object.

abstract connect_object()

Connect the object to DBus.

Handle emitted signals of the object with the _emit_signal method and handle incoming DBus calls with the _handle_call method.

abstract disconnect_object()

Disconnect the object from DBus.

Unregister the object and disconnect all signals.

property specification

DBus specification.

class GLibServer

Bases: object

The low-level DBus server library based on GLib.

classmethod emit_signal(*connection*, *object_path*, *interface_name*, *signal_name*, *parameters*, *destination=None*)

Emit a DBus signal.

classmethod get_call_info(*invocation*)

Get information about the DBus call.

Supported items:

sender str: The bus name that invoked the method

There can be more supported items in the future.

Parameters
invocation – an invocation of a DBus call

Returns
a dictionary of information about the DBus call

classmethod register_object(*connection*, *object_path*, *object_xml*, *callback*, *callback_args=()*)

Register an object on DBus.

classmethod set_call_error(*invocation*, *error_name*, *error_message*)

Set the error of the DBus call.

Parameters

- **invocation** – an invocation of a DBus call
- **error_name** – a DBus name of the error

- **error_message** – an error message

classmethod set_call_reply(*invocation*, *out_type*, *out_value*)

Set the reply of the DBus call.

Parameters

- **invocation** – an invocation of a DBus call
- **out_type** – a type of the reply
- **out_value** – a value of the reply

class ServerObjectHandler(*message_bus*, *object_path*, *obj*, *error_mapper=None*, *server=<class 'dasbus.server.handler.GLibServer'>*, *signal_factory=<class 'dasbus.signal.Signal'>*)

Bases: *AbstractServerObjectHandler*

The handler of an object published on DBus.

connect_object()

Connect the object to DBus.

disconnect_object()

Disconnect the object from DBus.

dasbus.server.interface module

accepts_additional_arguments(*method*)

Decorator for accepting extra arguments in a DBus method.

The decorator allows the server object handler to propagate additional information about the DBus call into the decorated method.

Use a dictionary of keyword arguments:

```
@accepts_additional_arguments
def Method(x: Int, y: Str, **info):
    pass
```

Or use keyword only parameters:

```
@accepts_additional_arguments
def Method(x: Int, y: Str, *, call_info):
    pass
```

At this moment, the library provides only the call_info argument generated by GLibServer.get_call_info, but the additional arguments can be customized in the _get_additional_arguments method of the server object handler.

Parameters

method – a DBus method

Returns

a DBus method with a flag

are_additional_arguments_supported(*method*)

Does the given DBus method accept additional arguments?

Parameters

method – a DBus method

Returns
True or False

dbus_class(*cls*)

DBus class.

A new DBus class can be defined as:

```
@dbus_class
class Class(Interface):
    ...
```

DBus class can implement DBus interfaces, but it cannot define a new interface.

The DBus XML specification will be generated from implemented interfaces (inherited) and it will be accessible as:

```
Class.__dbus_xml__
```

dbus_interface(*interface_name*, *namespace=()*)

DBus interface.

A new DBus interface can be defined as:

```
@dbus_interface
class Interface():
    ...
```

The interface will be generated from the given class cls with a name interface_name and added to the DBus XML specification of the class.

The XML specification is accessible as: .. code-block:: python

Interface.__dbus_xml__

It is conventional for member names on DBus to consist of capitalized words with no punctuation. The generator of the XML specification enforces this convention to prevent unintended changes in the specification. You can provide the XML specification yourself, or override the generator class to work around these constraints.

Parameters

- **interface_name** – a DBus name of the interface
- **namespace** – a sequence of strings

class dbus_signal(*definition=None*, *factory=<class 'dasbus.signal.Signal'>*)

Bases: `object`

DBus signal.

Can be used as:

```
Signal = dbus_signal()
```

Or as a method decorator:

```
@dbus_signal
def Signal(x: Int, y: Double):
    pass
```

Signal is defined by the type hints of a decorated method. This method is accessible as: signal.definition

If the signal is not defined by a method, it is expected to have no arguments and signal.definition is equal to None.

get_xml(*obj*)

Return XML specification of an object.

Parameters
obj – an object decorated with @dbus_interface or @dbus_class

Returns
a string with XML specification

dasbus.server.property module

exception PropertiesException

Bases: Exception

Exception for DBus properties.

class PropertiesInterface

Bases: object

Standard DBus interface org.freedesktop.DBus.Properties.

DBus objects don't have to inherit this class, because the DBus library provides support for this interface by default. This class only extends this support.

Report the changed property:

```
self.report_changed_property('X')
```

Emit all changes when the method is done:

```
@emits_properties_changed
def SetX(x: Int):
    self.set_x(x)
```

PropertiesChanged

DBus signal.

Can be used as:

```
Signal = dbus_signal()
```

Or as a method decorator:

```
@dbus_signal
def Signal(x: Int, y: Double):
    pass
```

Signal is defined by the type hints of a decorated method. This method is accessible as: signal.definition

If the signal is not defined by a method, it is expected to have no arguments and signal.definition is equal to None.

flush_changes()

Flush properties changes.

report_changed_property(*property_name*)

Reports changed DBus property.

Parameters
property_name – a name of a DBus property

emits_properties_changed(*method*)

Decorator for emitting properties changes.

The decorated method has to be a member of a class that inherits PropertiesInterface.

Parameters
method – a DBus method of a class that inherits PropertiesInterface

Returns
a wrapper of a DBus method that emits PropertiesChanged

dasbus.server.publishable module

class Publishable

Bases: object

Abstract class for Python objects that can be published on DBus.

Example:

```
# Define a publishable class.
class MyObject(Publishable):

    def for_publication(self):
        return MyDBusInterface(self)

# Create a publishable object.
my_object = MyObject()

# Publish the object on DBus.
DBus.publish_object("/org/project/x", my_object.for_publication())
```

abstract for_publication()

Return a DBus representation of this object.

Returns
an instance of @dbus_interface or @dbus_class

dasbus.server.template module

class BasicInterfaceTemplate(*implementation*)

Bases: object

Basic template for a DBus interface.

This template uses a software design pattern called proxy.

This class provides a recommended way how to define DBus interfaces and create publishable DBus objects. The class that defines a DBus interface should inherit this class and be decorated with @dbus_class or @dbus_interface decorator. The implementation of this interface will be provided by a separate object called

implementation. Therefore the methods of this class should call the methods of the implementation, the signals should be connected to the signals of the implementation and the getters and setters of properties should access the properties of the implementation.

```
@dbus_interface("org.myproject.X")
class InterfaceX(BasicInterfaceTemplate):
    def DoSomething(self) -> Str:
        return self.implementation.do_something()

class X(object):
    def do_something(self):
        return "Done!"

x = X()
i = InterfaceX(x)

DBus.publish_object("/org/myproject/X", i)
```

connect_signals()

Interconnect the signals.

You should connect the emit methods of the interface signals to the signals of the implementation. Every time the implementation emits a signal, this interface reemits the signal on DBus.

property implementation

Return the implementation of this interface.

Returns
an implementation

class InterfaceTemplate(*implementation*)

Bases: *BasicInterfaceTemplate*, *PropertiesInterface*

Template for a DBus interface.

The interface provides the support for the standard interface org.freedesktop.DBus.Properties.

Usage:

```
def connect_signals(self):
    super().connect_signals()
    self.implementation.module_properties_changed.connect(
        self.flush_changes
    )
    self.watch_property("X", self.implementation.x_changed)

@property
def X(self, x) -> Int:
    return self.implementation.x

@emits_properties_changed
def SetX(self, x: Int):
    self.implementation.set_x(x)
```

watch_property(*property_name*, *signal*)

Watch a DBus property.

Report a change when the property is changed.

Parameters

- **property_name** – a name of a DBus property
- **signal** – a signal that emits when the property is changed

Module contents

Submodules

dasbus.connection module

class AddressedMessageBus(*address*, **args*, ***kwargs*)

Bases: *MessageBus*

Representation of a connection for the specified address.

property address

The bus address.

class GLibConnection

Bases: object

The low-level DBus connection library based on GLib.

DEFAULT_FLAGS = <flags G_DBUS_CONNECTION_FLAGS_AUTHENTICATION_CLIENT | G_DBUS_CONNECTION_FLAGS_MESSAGE_BUS_CONNECTION of type Gio.DBusConnectionFlags>

static get_addressed_bus_connection(*bus_address*, *flags=<flags G_DBUS_CONNECTION_FLAGS_AUTHENTICATION_CLIENT | G_DBUS_CONNECTION_FLAGS_MESSAGE_BUS_CONNECTION of type Gio.DBusConnectionFlags>*, *observer=None*, *cancellable=None*)

Get a connection to a bus at the specified address.

static get_session_bus_connection(*cancellable=None*)

Get a session bus connection.

static get_system_bus_connection(*cancellable=None*)

Get a system bus connection.

class MessageBus(*error_mapper=None*, *provider=<class 'dasbus.connection.GLibConnection'>*)

Bases: AbstractMessageBus

Representation of a message bus based on D-Bus.

property connection

The DBus connection.

disconnect()

Disconnect from DBus.

get_proxy(*service_name*, *object_path*, *interface_name=None*, *proxy_factory=None*, ***proxy_arguments*)

Returns a proxy of a remote DBus object.

If the proxy factory is not specified, we will use a default one. If the interface name is set, we will choose InterfaceProxy, otherwise ObjectProxy.

If the interface name is set, we will add it to the additional arguments for the proxy factory.

Parameters

- **service_name** – a DBus name of a service
- **object_path** – a DBus path of an object
- **interface_name** – a DBus name of an interface or None
- **proxy_factory** – a factory of a DBus object proxy
- **proxy_arguments** – additional arguments for the proxy factory

Returns
a proxy object

property proxy

The proxy of DBus.

publish_object(*object_path*, *obj*, *server_factory=<class 'dasbus.server.handler.ServerObjectHandler'>*, ***server_arguments*)

Publish an object on DBus.

Parameters

- **object_path** – a DBus path of an object
- **obj** – an instance of @dbus_interface or @dbus_class
- **server_factory** – a factory of a DBus server object handler
- **server_arguments** – additional arguments for the server factory

register_service(*service_name*, *flags=1*)

Register a service on DBus.

Parameters

- **service_name** – a DBus name of a service
- **flags** – the flags argument of the RequestName DBus method

unpublish_object(*object_path*)

unregister_service(*object_path*)

class SessionMessageBus(*error_mapper=None*, *provider=<class 'dasbus.connection.GLibConnection'>*)

Bases: *MessageBus*

Representation of a session bus connection.

class SystemMessageBus(*error_mapper=None*, *provider=<class 'dasbus.connection.GLibConnection'>*)

Bases: *MessageBus*

Representation of a system bus connection.

dasbus.constants module

dasbus.error module

class AbstractErrorRule

Bases: object

Abstract rule for mapping a Python exception to a DBus error.

abstract get_name(*exception_type*)

Get a DBus name for the given exception type.

Parameters
exception_type – a type of the Python error

Returns
a name of the DBus error

abstract get_type(*error_name*)

Get an exception type of the given DBus error.

param error_name: a name of the DBus error :return: a type of the Python error

abstract match_name(*error_name*)

Is this rule matching the given DBus error?

Parameters
error_name – a name of the DBus error

Returns
True or False

abstract match_type(*exception_type*)

Is this rule matching the given exception type?

Parameters
exception_type – a type of the Python error

Returns
True or False

exception DBusError

Bases: Exception

A default DBus error.

class DefaultErrorRule(*default_type*, *default_namespace*)

Bases: *AbstractErrorRule*

Default rule for mapping a Python exception to a DBus error.

get_name(*exception_type*)

Get a DBus name for the given exception type.

get_type(*error_name*)

Get an exception type of the given DBus error.

match_name(*error_name*)

Is this rule matching the given DBus error?

match_type(*exception_type*)

Is this rule matching the given exception type?

class ErrorMapper

Bases: object

Class for mapping Python exceptions to DBus errors.

add_rule(*rule:* AbstractErrorRule)

Add a rule to the error mapper.

The new rule will have a higher priority than the rules already contained in the error mapper.

Parameters
rule (*an instance of AbstractErrorRule*) – an error rule

get_error_name(*exception_type*)

Get a DBus name of the Python exception.

Try to find a matching rule in the error mapper. If a rule matches the given exception type, use the rule to get the name of the DBus error.

The rules in the error mapper are processed in the reversed order to respect the priority of the rules.

Parameters
exception_type (*a subclass of Exception*) – a type of the Python error

Returns
a name of the DBus error

Raises
LookupError – if no name is found

get_exception_type(*error_name*)

Get a Python exception type of the DBus error.

Try to find a matching rule in the error mapper. If a rule matches the given name of a DBus error, use the rule to get the type of a Python exception.

The rules in the error mapper are processed in the reversed order to respect the priority of the rules.

Parameters
error_name – a name of the DBus error

Returns
a type of the Python exception

Return type
a subclass of Exception

Raises
LookupError – if no type is found

reset_rules()

Reset rules in the error mapper.

Reset the error rules to the initial state. All rules will be replaced with the default ones.

class ErrorRule(*exception_type*, *error_name*)

Bases: *AbstractErrorRule*

Rule for mapping a Python exception to a DBus error.

get_name(*exception_type*)

Get a DBus name for the given exception type.

get_type(*error_name*)

Get an exception type of the given DBus error.

match_name(*error_name*)

Is this rule matching the given DBus error?

match_type(*exception_type*)

Is this rule matching the given exception type?

get_error_decorator(*error_mapper*)

Generate a decorator for DBus errors.

Create a function for decorating Python exception classes. The decorator will add a new rule to the given error mapper that will map the class to the specified error name.

Definition of the decorator:

```
decorator(error_name, namespace=())
```

The decorator accepts a name of the DBus error and optionally a namespace of the DBus name. The namespace will be used as a prefix of the DBus name.

Usage:

```
# Create an error mapper.
error_mapper = ErrorMapper()

# Create a decorator for DBus errors and use it to map
# the class ExampleError to the name my.example.Error.
dbus_error = create_error_decorator(error_mapper)

@dbus_error("my.example.Error")
class ExampleError(DBusError):
    pass
```

Parameters

error_mapper – an error mapper

Returns

a decorator

dasbus.identifier module

class DBusInterfaceIdentifier(*namespace*, *basename=None*, *interface_version=None*)

Bases: DBusBaseIdentifier

Identifier of a DBus interface.

property interface_name

Full name of the DBus interface.

class DBusObjectIdentifier(*namespace*, *basename=None*, *interface_version=None*, *object_version=None*)

Bases: *DBusInterfaceIdentifier*

Identifier of a DBus object.

property object_path

Full path of the DBus object.

class DBusServiceIdentifier(*message_bus*, *namespace*, *basename=None*, *interface_version=None*, *object_version=None*, *service_version=None*)

Bases: *DBusObjectIdentifier*

Identifier of a DBus service.

get_proxy(*object_path=None*, *interface_name=None*, ***bus_arguments*)

Returns a proxy of the DBus object.

If no object path is specified, we will use the object path of this DBus service.

If no interface name is specified, we will use none and create a proxy from all interfaces of the DBus object.

Parameters

- **object_path** – an object identifier or a DBus path or None
- **interface_name** – an interface identifier or a DBus name or None
- **bus_arguments** – additional arguments for the message bus

Returns
a proxy object

property message_bus

Message bus of the DBus service.

Returns
a message bus

Return type
an instance of the MessageBus class

property service_name

Full name of a DBus service.

dasbus.loop module

class AbstractEventLoop

Bases: object

The abstract representation of the event loop.

It is necessary to run the event loop to handle emitted DBus signals or incoming DBus calls (in the DBus service).

Example:

```
# Create the event loop.
loop = EventLoop()

# Start the event loop.
loop.run()
```

(continues on next page)

(continued from previous page)

```
# Run loop.quit() to stop.
```

abstract quit()

Stop the event loop.

abstract run()

Start the event loop.

class EventLoop

Bases: *AbstractEventLoop*

The representation of the event loop.

quit()

Stop the event loop.

run()

Start the event loop.

dasbus.namespace module

get_dbus_name(**namespace*)

Create a DBus name from the given names.

Parameters
namespace – a sequence of names

Returns
a DBus name

get_dbus_path(**namespace*)

Create a DBus path from the given names.

Parameters
namespace – a sequence of names

Returns
a DBus path

get_namespace_from_name(*name*)

Return a namespace of the DBus name.

Parameters
name – a DBus name

Returns
a sequence of names

dasbus.signal module

class Signal

Bases: object

Default representation of a signal.

connect(*callback*)

Connect to a signal.

Parameters
callback – a function to register

disconnect(*callback=None*)

Disconnect from a signal.

If no callback is specified, then all functions will be unregistered from the signal.

If the specified callback isn't registered, do nothing.

Parameters
callback – a function to unregister or None

emit(**args*, ***kwargs*)

Emit a signal with the given arguments.

dasbus.specification module

class DBusSpecification

Bases: object

DBus XML specification.

ACCESS_READ = 'read'

ACCESS_READWRITE = 'readwrite'

ACCESS_WRITE = 'write'

DIRECTION_IN = 'in'

DIRECTION_OUT = 'out'

class Method(*name*, *interface_name*, *in_type*, *out_type*)

Bases: tuple

in_type

Alias for field number 2

interface_name

Alias for field number 1

name

Alias for field number 0

out_type

Alias for field number 3

class Property(*name*, *interface_name*, *readable*, *writable*, *type*)

Bases: tuple

interface_name

Alias for field number 1

name

Alias for field number 0

readable

Alias for field number 2

type

Alias for field number 4

writable

Alias for field number 3

RETURN_PARAMETER = 'return'

STANDARD_INTERFACES = '\n <node>\n <interface
name="org.freedesktop.DBus.Introspectable">\n <method name="Introspect">\n <arg
type="s" name="xml_data" direction="out"/>\n </method>\n </interface>\n <interface
name="org.freedesktop.DBus.Peer">\n <method name="Ping"/>\n <method
name="GetMachineId">\n <arg type="s" name="machine_uuid" direction="out"/>\n
</method>\n </interface>\n <interface name="org.freedesktop.DBus.Properties">\n
<method name="Get">\n <arg type="s" name="interface_name" direction="in"/>\n <arg
type="s" name="property_name" direction="in"/>\n <arg type="v" name="value"
direction="out"/>\n </method>\n <method name="GetAll">\n <arg type="s"
name="interface_name" direction="in"/>\n <arg type="a{sv}" name="properties"
direction="out"/>\n </method>\n <method name="Set">\n <arg type="s"
name="interface_name" direction="in"/>\n <arg type="s" name="property_name"
direction="in"/>\n <arg type="v" name="value" direction="in"/>\n </method>\n <signal
name="PropertiesChanged">\n <arg type="s" name="interface_name"/>\n <arg
type="a{sv}" name="changed_properties"/>\n <arg type="as"
name="invalidated_properties"/>\n </signal>\n </interface>\n </node>\n '

class Signal(*name*, *interface_name*, *type*)

Bases: tuple

interface_name

Alias for field number 1

name

Alias for field number 0

type

Alias for field number 2

add_member(*member*)

Add a member of a DBus interface.

classmethod from_xml(*xml*)

Return a DBus specification for the given XML.

get_member(*interface_name*, *member_name*)

Get a member of a DBus interface.

property interfaces

Interfaces of the DBus specification.

property members

Members of the DBus specification.

exception DBusSpecificationError

Bases: Exception

Exception for the DBus specification errors.

class DBusSpecificationParser

Bases: object

Class for parsing DBus XML specification.

classmethod parse_specification(*xml*, *factory=<class 'dasbus.specification.DBusSpecification'>*)

Generate a representation of a DBus XML specification.

Parameters

- **xml** – the XML specification to parse
- **factory** – the DBus specification factory

Returns

a representation od the DBus specification

xml_parser

alias of *XMLParser*

dasbus.structure module

class DBusData

Bases: object

Object representation of data in a DBus structure.

Classes derived from this class should represent specific types of DBus structures. They will support a conversion from a DBus structure of this type to a Python object and back.

classmethod from_structure(*structure: Dict[str,* Variant*]*)

Convert a DBus structure to a data object.

Parameters

structure – a DBus structure

Returns

a data object

classmethod from_structure_list(*structures: List[Dict[str,* Variant*]]*)

Convert DBus structures to data objects.

Parameters

structures – a list of DBus structures

Returns

a list of data objects

classmethod to_structure(*data*) → Dict[str, *Variant*]

Convert this data object to a DBus structure.

Returns
a DBus structure

classmethod to_structure_list(*objects*) → List[Dict[str, *Variant*]]

Convert data objects to DBus structures.

Parameters
objects – a list of data objects

Returns
a list of DBus structures

exception DBusStructureError

Bases: Exception

General exception for DBus structure errors.

compare_data(*obj*, *other*)

Compare data of the given data objects.

Parameters

- **obj** – a data object
- **other** – another data object

Returns
True if the data is equal, otherwise False

generate_string_from_data(*obj*, *skip=None*, *add=None*)

Generate a string representation of a data object.

Set the argument 'skip' to skip attributes with sensitive data.

Set the argument 'add' to add other values to the string representation. The attributes in the string representation will be sorted alphabetically.

Parameters

- **obj** – a data object
- **skip** – a list of names that should be skipped or None
- **add** – a dictionary of attributes to add or None

Returns
a string representation of the data object

dasbus.typing module

Bool

alias of bool

Double

alias of float

Int

alias of int

Str

alias of `str`

class Variant(*format_string*, *value*)

Bases: `Variant`

Constructors

```
new_array(child_type:GLib.VariantType=None, children:list=None) -> GLib.Variant
new_boolean(value:bool) -> GLib.Variant
new_byte(value:int) -> GLib.Variant
new_bytestring(string:list) -> GLib.Variant
new_bytestring_array(strv:list) -> GLib.Variant
new_dict_entry(key:GLib.Variant, value:GLib.Variant) -> GLib.Variant
new_double(value:float) -> GLib.Variant
new_fixed_array(element_type:GLib.VariantType, elements=None, n_elements:int,␣
↪element_size:int) -> GLib.Variant
new_from_bytes(type:GLib.VariantType, bytes:GLib.Bytes, trusted:bool) -> GLib.
↪Variant
new_from_data(type:GLib.VariantType, data:list, trusted:bool, notify:GLib.
↪DestroyNotify, user_data=None) -> GLib.Variant
new_handle(value:int) -> GLib.Variant
new_int16(value:int) -> GLib.Variant
new_int32(value:int) -> GLib.Variant
new_int64(value:int) -> GLib.Variant
new_maybe(child_type:GLib.VariantType=None, child:GLib.Variant=None) -> GLib.Variant
new_object_path(object_path:str) -> GLib.Variant
new_objv(strv:list) -> GLib.Variant
new_signature(signature:str) -> GLib.Variant
new_string(string:str) -> GLib.Variant
new_strv(strv:list) -> GLib.Variant
new_tuple(children:list) -> GLib.Variant
new_uint16(value:int) -> GLib.Variant
new_uint32(value:int) -> GLib.Variant
new_uint64(value:int) -> GLib.Variant
new_variant(value:GLib.Variant) -> GLib.Variant
```

get_string(*self*) → str, length:int

keys()

static new_tuple(*children: list*) → *GLib.Variant*

classmethod split_signature(*signature*)

Return a list of the element signatures of the topmost signature tuple.

If the signature is not a tuple, it returns one element with the entire signature. If the signature is an empty tuple, the result is [].

This is useful for e. g. iterating over method parameters which are passed as a single Variant.

unpack()

Decompose a GVariant into a native Python object.

class VariantType(***kwargs*)

Bases: `Boxed`

Constructors

```
new(type_string:str) -> GLib.VariantType
new_array(element:GLib.VariantType) -> GLib.VariantType
new_dict_entry(key:GLib.VariantType, value:GLib.VariantType) -> GLib.VariantType
new_maybe(element:GLib.VariantType) -> GLib.VariantType
new_tuple(items:list) -> GLib.VariantType
```

checked_ = gi.FunctionInfo(checked_)

copy = gi.FunctionInfo(copy)

dup_string = gi.FunctionInfo(dup_string)

element = gi.FunctionInfo(element)

equal = gi.FunctionInfo(equal)

first = gi.FunctionInfo(first)

free = gi.FunctionInfo(free)

get_string_length = gi.FunctionInfo(get_string_length)

hash = gi.FunctionInfo(hash)

is_array = gi.FunctionInfo(is_array)

is_basic = gi.FunctionInfo(is_basic)

is_container = gi.FunctionInfo(is_container)

is_definite = gi.FunctionInfo(is_definite)

is_dict_entry = gi.FunctionInfo(is_dict_entry)

is_maybe = gi.FunctionInfo(is_maybe)

is_subtype_of = gi.FunctionInfo(is_subtype_of)

is_tuple = gi.FunctionInfo(is_tuple)

is_variant = gi.FunctionInfo(is_variant)

key = gi.FunctionInfo(key)

n_items = gi.FunctionInfo(n_items)

new = gi.FunctionInfo(new)

new_array = gi.FunctionInfo(new_array)

new_dict_entry = gi.FunctionInfo(new_dict_entry)

new_maybe = gi.FunctionInfo(new_maybe)

new_tuple = gi.FunctionInfo(new_tuple)

next = gi.FunctionInfo(next)

string_get_depth_ = gi.FunctionInfo(string_get_depth_)

string_is_valid = gi.FunctionInfo(string_is_valid)

string_scan = gi.FunctionInfo(string_scan)

value = gi.FunctionInfo(value)

class VariantUnpacker

Bases: *VariantUnpacking*

Class for unpacking variants.

classmethod apply(*variant*)

Unpack the specified variant.

Parameters
variant – a variant to unpack

Returns
an unpacked value

class VariantUnpacking

Bases: `object`

Set of functions of unpacking a variant.

This class is doing the same as the unpack method of the Variant class, but it allows to reuse the code for other variant modifications.

class VariantUnwrapper

Bases: *VariantUnpacking*

Class for unwrapping variants.

classmethod apply(*variant*)

Unwrap the specified variant.

Parameters
variant – a variant to unwrap

Returns
a unwrapped value

get_dbus_type(*type_hint*)

Return DBus representation of a type hint.

Parameters
type_hint – a type hint

Returns
a string with DBus representation

get_native(*value*)

Decompose a DBus value into a native Python object.

This function is useful for testing, when the DBus library doesn't decompose arguments and return values of DBus calls.

Parameters
value – a DBus value

Returns
a native Python object

`get_type_arguments`(*type_hint*)

Get the arguments of the type hint.

For example, Str and Int are arguments of the type hint Tuple(Str, Int).

Parameters
type_hint – a type hint

Returns
a type arguments

`get_variant`(*type_hint*, *value*)

Return a variant data type.

The type of a variant is specified with a type hint.

Example:

```
v1 = get_variant(Bool, True)
v2 = get_variant(List[Int], [1,2,3])
```

Parameters

- **type_hint** – a type hint or a type string
- **value** – a value of the variant

Returns
an instance of Variant

`get_variant_type`(*type_hint*)

Return a type of a variant data type.

Parameters
type_hint – a type hint or a type string

Returns
an instance of VariantType

`is_base_type`(*type_hint*, *base_type*)

Is the given base type a base of the specified type hint?

For example, List is a base of the type hint List[Int] and Int is a base of the type hint Int. A class is a base of itself and of every subclass of this class.

Parameters

- **type_hint** – a type hint
- **base_type** – a base type

Returns
True or False

`is_tuple_of_one`(*type_hint*)

Is the type hint a tuple of one item?

Parameters
type_hint – a type hint or a type string

Returns
True or False

`unwrap_variant`(*variant*)

Unwrap a variant data type.

Unlike the unpack method of the Variant class, this function doesn't recursively unpacks all variants in the data structure. It will unpack only the topmost variant.

The implementation is inspired by the unpack method.

Parameters
variant – a variant

Returns
a value

dasbus.unix module

`class GLibClientUnix`

Bases: *GLibClient*

The low-level DBus client library based on GLib.

`classmethod async_call`(*connection*, *service_name*, *object_path*, *interface_name*, *method_name*, *parameters*, *reply_type*, *callback*, *callback_args=()*, *flags=0*, *timeout=2147483647*)

Asynchronously call a DBus method.

`classmethod sync_call`(*connection*, *service_name*, *object_path*, *interface_name*, *method_name*, *parameters*, *reply_type*, *flags=0*, *timeout=2147483647*)

Synchronously call a DBus method.

Returns
a result of the DBus call

`class GLibServerUnix`

Bases: *GLibServer*

The low-level DBus server library based on GLib.

Adds Unix FD Support to base class

`classmethod emit_signal`(*connection*, *object_path*, *interface_name*, *signal_name*, *parameters*, *destination=None*)

Emit a DBus signal.

GLib doesn't seem to support Unix file descriptors in signals. Swap Unix file descriptors with indexes into a list of Unix file descriptors, but emit just the indexes. Log a warning to inform users about the limited support.

`classmethod set_call_reply`(*invocation*, *out_type*, *out_value*)

Set the reply of the DBus call.

dasbus.xml module

class XMLGenerator

Bases: *XMLParser*

Class for generating XML.

static add_child(*parent_element*, *child_element*)

Append the child element to the parent element.

static add_comment(*element*, *comment*)

static create_interface(*name*)

Create an interface element.

static create_method(*name*)

Create a method element.

static create_node()

Create a node element called node.

static create_parameter(*name*, *param_type*, *direction*)

Create a parameter element.

static create_property(*name*, *property_type*, *access*)

Create a property element.

static create_signal(*name*)

Create a signal element.

static element_to_xml(*element*)

Return XML of the element.

static prettify_xml(*xml*)

Return pretty printed normalized XML.

Python 3.8 changed the order of the attributes and introduced the function canonicalize that should be used to normalize XML.

class XMLParser

Bases: `object`

Class for parsing XML.

static get_access(*node*)

static get_direction(*node*)

static get_interfaces_from_node(*node_element*)

Return a dictionary of interfaces defined in a node element.

static get_name(*node*)

static get_type(*node*)

static has_name(*node*, *node_name*)

static is_interface(*member_node*)

static is_member(*member_node*)

static is_method(*member_node*)

static is_parameter(*member_node*)

static is_property(*member_node*)

static is_signal(*member_node*)

static xml_to_element(*xml*)

Module contents

2.5.4 Examples

Look at the complete examples or DBus services of the Anaconda Installer for more inspiration.

Basic usage

Show the current hostname.

```
from dasbus.connection import SystemMessageBus
bus = SystemMessageBus()

proxy = bus.get_proxy(
    "org.freedesktop.hostname1",
    "/org/freedesktop/hostname1"
)

print(proxy.Hostname)
```

Send a notification to the notification server.

```
from dasbus.connection import SessionMessageBus
bus = SessionMessageBus()

proxy = bus.get_proxy(
    "org.freedesktop.Notifications",
    "/org/freedesktop/Notifications"
)

id = proxy.Notify(
    "", 0, "face-smile", "Hello World!",
    "This notification can be ignored.",
    [], {}, 0
)

print("The notification {} was sent.".format(id))
```

Handle a closed notification.

```
from dasbus.loop import EventLoop
loop = EventLoop()

from dasbus.connection import SessionMessageBus
bus = SessionMessageBus()

proxy = bus.get_proxy(
    "org.freedesktop.Notifications",
    "/org/freedesktop/Notifications"
)

def callback(id, reason):
    print("The notification {} was closed.".format(id))

proxy.NotificationClosed.connect(callback)
loop.run()
```

Asynchronously fetch a list of network devices.

```
from dasbus.loop import EventLoop
loop = EventLoop()

from dasbus.connection import SystemMessageBus
bus = SystemMessageBus()

proxy = bus.get_proxy(
    "org.freedesktop.NetworkManager",
    "/org/freedesktop/NetworkManager"
)

def callback(call):
    print(call())

proxy.GetDevices(callback=callback)
loop.run()
```

Define the org.example.HelloWorld service.

```
class HelloWorld(object):
    __dbus_xml__ = """
    <node>
        <interface name="org.example.HelloWorld">
            <method name="Hello">
                <arg direction="in" name="name" type="s" />
                <arg direction="out" name="return" type="s" />
            </method>
        </interface>
    </node>
    """

    def Hello(self, name):
        return "Hello {}!".format(name)
```

Define the org.example.HelloWorld service with an automatically generated XML specification.

```
from dasbus.server.interface import dbus_interface
from dasbus.typing import Str

@dbus_interface("org.example.HelloWorld")
class HelloWorld(object):

    def Hello(self, name: Str) -> Str:
        return "Hello {}!".format(name)

print(HelloWorld.__dbus_xml__)
```

Publish the org.example.HelloWorld service on the session message bus.

```
from dasbus.connection import SessionMessageBus
bus = SessionMessageBus()
bus.publish_object("/org/example/HelloWorld", HelloWorld())
bus.register_service("org.example.HelloWorld")

from dasbus.loop import EventLoop
loop = EventLoop()
loop.run()
```

Support for Unix file descriptors

The support for Unix file descriptors is disabled by default. It needs to be explicitly enabled when you create a DBus proxy or publish a DBus object that could send or receive Unix file descriptors.

> **Warning:** This functionality is supported only on UNIX.

Send and receive Unix file descriptors with a DBus proxy.

```
import os
from dasbus.connection import SystemMessageBus
from dasbus.unix import GLibClientUnix
bus = SystemMessageBus()

proxy = bus.get_proxy(
    "org.freedesktop.login1",
    "/org/freedesktop/login1",
    client=GLibClientUnix
)

fd = proxy.Inhibit(
    "sleep", "my-example", "Running an example", "block"
)

proxy.ListInhibitors()
os.close(fd)
```

Allow to send and receive Unix file descriptors within the /org/example/HelloWorld DBus object.

```
from dasbus.unix import GLibServerUnix
bus.publish_object(
    "/org/example/HelloWorld",
    HelloWorld(),
    server=GLibServerUnix
)
```

Management of DBus names and paths

Use constants to define DBus services and objects.

```
from dasbus.connection import SystemMessageBus
from dasbus.identifier import DBusServiceIdentifier, DBusObjectIdentifier

NETWORK_MANAGER_NAMESPACE = (
    "org", "freedesktop", "NetworkManager"
)

NETWORK_MANAGER = DBusServiceIdentifier(
    namespace=NETWORK_MANAGER_NAMESPACE,
    message_bus=SystemMessageBus()
)

NETWORK_MANAGER_SETTINGS = DBusObjectIdentifier(
    namespace=NETWORK_MANAGER_NAMESPACE,
    basename="Settings"
)
```

Create a proxy of the org.freedesktop.NetworkManager service.

```
proxy = NETWORK_MANAGER.get_proxy()
print(proxy.NetworkingEnabled)
```

Create a proxy of the /org/freedesktop/NetworkManager/Settings object.

```
proxy = NETWORK_MANAGER.get_proxy(NETWORK_MANAGER_SETTINGS)
print(proxy.Hostname)
```

See a complete example.

Error handling

Use exceptions to propagate and handle DBus errors. Create an error mapper and a decorator for mapping Python exception classes to DBus error names.

```
from dasbus.error import ErrorMapper, DBusError, get_error_decorator
error_mapper = ErrorMapper()
dbus_error = get_error_decorator(error_mapper)
```

Use the decorator to register Python exceptions that represent DBus errors. These exceptions can be raised by DBus services and caught by DBus clients in the try-except block.

```
@dbus_error("org.freedesktop.DBus.Error.InvalidArgs")
class InvalidArgs(DBusError):
    pass
```

The message bus will use the specified error mapper to automatically transform Python exceptions to DBus errors and back.

```
from dasbus.connection import SessionMessageBus
bus = SessionMessageBus(error_mapper=error_mapper)
```

See a complete example.

Timeout for a DBus call

Call DBus methods with a timeout (specified in milliseconds).

```
proxy = NETWORK_MANAGER.get_proxy()

try:
    proxy.CheckConnectivity(timeout=3)
except TimeoutError:
    print("The call timed out!")
```

Support for DBus structures

Represent DBus structures by Python objects. A DBus structure is a dictionary of attributes that maps attribute names to variants with attribute values. Use Python objects to define such structures. They can be easily converted to a dictionary, send via DBus and converted back to an object.

```
from dasbus.structure import DBusData
from dasbus.typing import Str, get_variant

class UserData(DBusData):
    def __init__(self):
        self._name = ""

    @property
    def name(self) -> Str:
        return self._name

    @name.setter
    def name(self, name):
        self._name = name

data = UserData()
data.name = "Alice"

print(UserData.to_structure(data))
print(UserData.from_structure({
    "name": get_variant(Str, "Bob")
}))
```

See a complete example.

Management of dynamic DBus objects

Create Python objects that can be automatically published on DBus. These objects are usually managed by DBus containers and published on demand.

```
from dasbus.server.interface import dbus_interface
from dasbus.server.template import InterfaceTemplate
from dasbus.server.publishable import Publishable
from dasbus.typing import Str

@dbus_interface("org.example.Chat")
class ChatInterface(InterfaceTemplate):

    def Send(self, message: Str):
        return self.implementation.send()

class Chat(Publishable):

    def for_publication(self):
        return ChatInterface(self)

    def send(self, message):
        print(message)
```

Use DBus containers to automatically publish dynamically created Python objects. A DBus container converts publishable Python objects into DBus paths and back. It generates unique DBus paths in the specified namespace and assigns them to objects. Each object is published when its DBus path is requested for the first time.

```
from dasbus.connection import SessionMessageBus
from dasbus.server.container import DBusContainer

container = DBusContainer(
    namespace=("org", "example", "Chat"),
    message_bus=SessionMessageBus()
)

print(container.to_object_path(Chat()))
```

See a complete example.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

E

F

G

H

I

K

M

N

O

P

Q

R

S

T

U

V

W

X