Bluetit

D-Bus controlled system daemon providing full connectivity to **AirVPN** and OpenVPN servers

System Architecture and D-Bus Interface Specifications

Developer's Reference Manual

Abstract

AirVPN–SUITE is a collection of applications, designed and developed by AirVPN, providing VPN connectivity both to AirVPN servers and to generic OpenVPN systems. The core component of the Suite is Bluetit, a lightweight D-Bus controlled system daemon and providing VPN connectivity through OpenVPN 3 AirVPN, a fork from the original OpenVPN 3 branch.

Bluetit exposes a D-Bus interface which can be used by client applications in order to control the daemon and also providing full interaction and connectivity with the whole AirVPN infrastructure.

Document Version: 1.0 2021-08-03
Revision: 1
Bluetit Version: 1.1.0
Release Date: 3 August 2021
Author: promind
Draft Reviser: pj

Typeset with LaTeX 2ε using Latin Modern font family and based on Donald Knuth's Computer Modern font family created with METAFONT

Contents

List of Figures

List of Coding Examples

Introduction

AirVPN–SUITE[1] is a free and open source set of tools and applications, designed and developed by AirVPN, providing VPN connectivity both to AirVPN servers and generic OpenVPN systems, targeting Linux[2] distributions[3] and, partially, macOS™[4] systems.

The core component of the Suite is Bluetit, a lightweight D-Bus controlled system daemon providing VPN connectivity through OpenVPN3-AirVPN, a fork from the original OpenVPN 3 branch.

Bluetit exposes a D-Bus interface which can be used by client applications in order to control the daemon and provide full interaction and connectivity with the whole AirVPN infrastructure.

Full user documentation for AirVPN–SUITE is available at `https://airvpn.org/suite/readme` whereas the complete source code is available at AirVPN–SUITE GitLab repository at `https://gitlab.com/AirVPN/AirVPN-Suite`.

The goal of the AirVPN–SUITE is to support the widest range of Linux distributions and architectures, as well as the currently supported macOS architectures. The suite provides connectivity and support to any OpenVPN server – including AirVPN servers – by using the OpenVPN3-AirVPN fork[5].

The OpenVPN3-AirVPN fork is actively maintained and developed by AirVPN and derived from the master branch of OpenVPN3[6].

From a design point of view, the AirVPN–SUITE is a set of classes, some of them interdependent one to each other, providing full access and interoperability with the AirVPN family server located all over the world and allowing its users to automatically connect to the best VPN server according to traffic and connectivity at a given time and country.

Connectivity to AirVPN servers is provided by the OpenVPN3-AirVPN class library, derived from the master branch of OpenVPN3 project. The OpenVPN3-AirVPN class library, besides fixing some bugs and unexpected behaviors found in the main branch, it also adds new features to the project, including those allowing OpenVPN3 to get a better performance and integration with AirVPN servers.

AirVPN–SUITE is currently made of three major components (applications):

- Bluetit, a distributed and lightweight system daemon based on D-Bus[7] and currently available for Linux systems only
- `Goldcrest`, a command line client for Bluetit
- `Hummingbird`, a stand-alone and lightweight application based on OpenVPN3 and available both for Linux and macOS systems

Bluetit – and therefore its client `Goldcrest` as well as any other client – are fully based on D-Bus for their inter-process communication needs, while `Hummingbird`, being a standalone application, can be virtually ported to any development system or environment fully compliant to standard C++ 14. At the time of writing this document, OpenVPN3-AirVPN provides native support for Linux, macOS and Windows®[8] only.

It should however be noted that all the classes and tools designed and developed for the AirVPN–SUITE and not depending on other tools and libraries (such as D-Bus) have been designed by using standard C++ 11 classes and convention, therefore ensuring the highest level of portability and use in the widest

[1] AirVPN–SUITE can be freely downloaded from AirVPN's website at `https://airvpn.org/linux/suite`

[2] Linux Kernel is a free open source Unix-like operating system kernel and released under the GNU GPL2 license. `https://www.kernel.org`

[3] A Linux distribution is a packaged set of tools and software, including the Linux kernel, system software and libraries, many of them being provided by the GNU Project.

[4] Mac™ and macOS™ are trademarks of Apple Inc., registered in the U.S. and other countries and regions.

[5] `https://github.com/AirVPN/openvpn3-airvpn`

[6] OpenVPN and OpenVPN3 are Copyright © 2002-2021 OpenVPN Inc. `https://github.com/OpenVPN/openvpn3`

[7] D-Bus is an inter-process communication (IPC) mechanism and part of the freedesktop.org project `https://www.freedesktop.org/wiki/Software/dbus/`

[8] Windows® is a registered trademark of Microsoft Corp.

number of systems, compilers and environments. The AirVPN–SUITE also makes use of a small set of libraries part of the GNU Project[9] which are commonly found and available in every Linux Distribution and, thanks to their highly portable design, are available for other systems as well, including macOS.

This document covers Bluetit infrastructure and architecture as well as providing a complete reference for all the AirVPN's classes on which the suite is based. The goal is to give any developer who wishes to write a Bluetit client or a tool providing AirVPN interconnectivity, a complete reference about the internals of both Bluetit daemon and the AirVPN–SUITE C++ classes.

The document is structured in chapters, giving an overview and in-depth information about Bluetit architecture and how a client should communicate with it in order to request and get all the available services provided by the daemon. The preferred method of inter-operating with Bluetit is by using AirVPN–SUITE C++ classes, although this is limiting the development of a client in C++ only.

All the AirVPN–SUITE classes can be virtually ported to any object-oriented programming language provided it can offer access or support to D-Bus. Also note AirVPN–SUITE classes are based on D-Bus low level C API and the use of an object-oriented programming language is not mandatory provided the target functions and/or classes are developed according to the AirVPN–SUITE classes marshaling mechanism, which is essential for the whole architecture in order to exchange data to and from the clients and the daemon.

[9] "GNU's Not Unix" home page `https://www.gnu.org/`

Chapter 1

Bluetit System Architecture

Bluetit is the core component of the AirVPN–SUITE and it is running in the system as a real *daemon*[1]. This basically means the Bluetit daemon cannot be directly controlled by using a terminal – as a matter of fact, it cannot acquire any terminal control – save the case of sending it signals corresponding to specific actions or tasks.

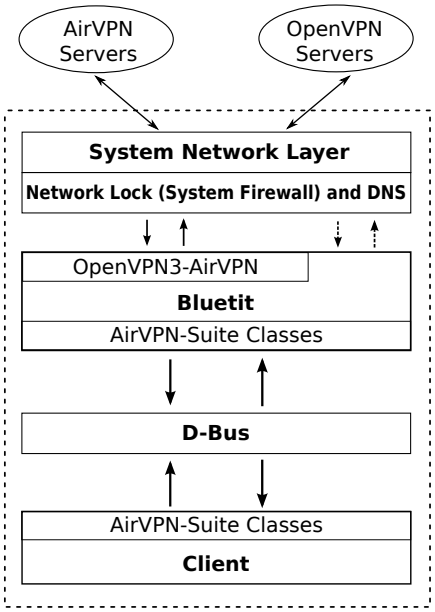

Figure 1.1: Bluetit client/server architecture

The only allowed way to control Bluetit daemon is by using an authorized client started from an authorized user which sends permitted methods through the D-Bus inter-process communication daemon.

Bluetit uses a simple client/server architecture and the inter-process communication between the two parts is realized with the D-Bus daemon running in the system. Bluetit, in turn, reaches the *outside world* – that is, OpenVPN servers, either run by AirVPN or other parties – by using OpenVPN3-AirVPN C++ class library which is a direct fork of the OpenVPN3 master branch. The connection with the external VPN servers is in charge of the system network layer sending and receiving data through a dedicated and encrypted IP tunnel[2] created and managed by OpenVPN3.

Figure 1.1 shows Bluetit architecture and how it interacts with the system components and resources. Bluetit is the main and central component of the architecture and controls the communication both with D-Bus and the network. The inter-process communication with D-Bus is ensured by the AirVPN–SUITE C++ classes and taking care of the whole process, from receiving and sending data to and from the client up to providing a marshaling mechanism capable of transparently handling complex datasets. This architecture is therefore effective in representing a virtually endless set of information and with a variable and non-homogeneous structure. This includes – but it is not limited to – objects belonging to the same class as well as unrelated objects or data structures.

The network part of Bluetit actually is a bit more complex as it needs to take care of different aspects in order to provide a secure and private VPN connection, as well as preventing data and DNS[3] leaks, according to its configuration or client needs.

The internal architecture of Bluetit is made of components each taking care and managing specific

[1] In Unix-like operating systems, a daemon is a program running as a background process, usually created by *forking* a child process and then immediately exiting, therefore ensuring the `init` process to adopt it as a child process. Moreover and for security reasons, a real daemon must be dissociated from any `TTY` terminal – including closing all standard I/O streams – as well as forcing and guaranteeing the child (forked) process to not be the session leader, therefore preventing the daemon from ever acquiring a controlling terminal.

[2] An IP tunnel is a dedicated Internet Protocol (IP) network communications channel between two networks and the exchange of data is realized by encapsulating data packets. Specifically, in case of Bluetit, the packets are encrypted before sending and decrypted upon reception, thus ensuring a secure and private communication

[3] DNS is the acronym for "Domain Name System", a hierarchical and decentralized naming system allowing a computer or device connected to the Internet to properly resolve and get the real IP address of a FQDN (fully qualified domain name) or URI (Universal Resource Identifier) such as `https://www.airvpn.org`

services. These components are inter-connected and may depend one on each other or however their scope and service may be relative to other ones.

1.1 Bluetit Components

Bluetit, the core component of AirVPN–SUITE, is built as a monolithic daemon and it is made of six major and inter-connected components which can be summarized like this:

- D-Bus layer
- Options and command manager
- OpenVPN3 service
- AirVPN manager
- Network Lock and DNS manager
- Logger

The above components are usually implemented by a specific C++ class or a set of classes, therefore taking advantage of the object-oriented analysis and design paradigm (OOAD) optionally ensuring the portability of the project or part of its components. The choice of having Bluetit as a monolithic component encapsulating other components is dictated by the need of not calling or loading external components or modules at run time, save the system libraries and tools essential for Network Lock and DNS management. However, also in this case, the use of external tools is minimized and always called by using a "best effort practice" in order to ensure a reliable security model while trying to prevent and limit external exploits as much as possible.

1.1.1 D-Bus Layer

The D-Bus layer takes care of inter-process communication between the Bluetit daemon and client by using D-Bus low level API provided by `libdbus`, a library part of the D-Bus installation set, available in any Linux system using and having a fully installed and configured D-Bus.

Specifically, Bluetit's D-Bus Layer waits for any incoming D-Bus method, executes the corresponding action and then replies to the client by communicating the operation status. For more information about Bluetit's D-Bus methods and use, refer to section 5.4 Public D-Bus Methods.

Although Bluetit D-Bus layer is using low level C API functions provided by `libdbus`, the high level interaction between client and server is realized by specialized and custom classes specifically built for the AirVPN–SUITE and part of it. These classes implement a handy wrapper for low level D-Bus C API functions, therefore allowing a more linear, consistent and homogeneous integration with C++. Although the D-Bus connection between Bluetit daemon and its client could preferably be realized by using AirVPN–SUITE's D-Bus classes, the developer is not forced to do so and every D-Bus library or facility can be virtually used to accomplish this task.

In case the developer wishes to use a different D-Bus library or functions to communicate with Bluetit, it is mandatory for the system to obey and provide the same marshaling mechanism implemented in AirVPN–SUITE D-Bus classes. The information exchanged to and from Bluetit can be quite complex and representing unrelated data sets, structures and objects in the very same transaction. The marshaling mechanism implemented in the AirVPN–SUITE D-Bus classes can transparently process these data and present them to the upper class in the form of C++ data, structures and objects. The AirVPN–SUITE D-Bus classes are explained and covered in detail in chapters 2, 3 and 4.

The marshaling mechanism implemented by AirVPN–SUITE D-Bus classes will not be covered in this document as it goes beyond of its scope. Those who are interested in implementing a similar marshaling mechanism in their custom clients can refer to the source code of `DBusResponse` class available in the official AirVPN–SUITE repository[4], in particular `fromString()` and `toString()` methods.

Bluetit D-Bus layer needs to be properly configured in order to ensure a safe and controlled access and communication. This is done by adding specific unit files to the D-Bus configuration, both for the server daemon and the client, as explained in chapter 5 Bluetit D-Bus Interface.

The AirVPN–SUITE distribution package includes *ready-to-use* D-Bus unit files which are installed during the installation process. The system administrator is free to change, set and fine tune the access

[4] AirVPN–SUITE GitLab repository is avilable at `https://gitlab.com/AirVPN/AirVPN-Suite`

policy defined in these unit files in order to suit local system needs. By default, D-Bus access is granted only to users belonging to the system user group `airvpn` and, of course, the `root` user.

The use of Bluetit D-Bus interface is granted on an exclusive policy. This does mean Bluetit accepts one client at a time only, that is the client successfully connecting to the interface and therefore taking the exclusive use of it until it closes the connection or Bluetit decides to terminate the session.

The exclusive policy is used in order to prevent external interference from other clients and which could deeply affect and change the current session status. In this specific case, a session is anything started or requested by the client to the Bluetit daemon, such as information about an AirVPN server or starting a VPN connection.

A session can terminate upon client request or when Bluetit terminates a task requested by the client. For example, in case a `SIGTERM` signal is sent to the client, it subsequently requests to Bluetit the disconnection from the current VPN server. In this case it is the client which asks for the termination of the session, although it is Bluetit to actually terminate it.

Another case is when it is Bluetit to decide the session is terminated at the completion of task associated to the client's request and, after having sent the response to it, an end of session event will follow.

A session internally started by Bluetit – such as connecting to an AirVPN server at boot time – is different from a client session and, as a matter of fact, it can be controlled by any authorized client by starting a dedicated session in order to alter Bluetit's internal session status, such as pausing or disconnecting the active VPN connection.

The `root` user can change the status of Bluetit at any time – including terminating a session, whether internal or started by a client – by sending system signals to it or by using system administration tools, such as `systemd` commands.

1.1.2 Options and Command Manager

Bluetit is a daemon driven by commands, options and system signals. While the management of system signals is implemented through the relative standard C library functions (such as `signal()`) and dedicated handlers, the options and commands are entirely managed by a specific layer.

Each function, command or service provided by Bluetit to the client must be requested by sending the corresponding option or D-Bus method. The structure and management of Bluetit commands and options is similar to the standard C mechanism used to pass options to a program from the terminal or a shell script. In other words, it is similar to the concept adopted in the well-known `argc` and `argv` arguments used by the `main()` function in a C program.

In the specific case of Bluetit, the *array* of options and values are represented by a standard C++ vector of strings (that is, a `std::vector<std::string>` object) and passed from the client to the server through the inter-process communication service of D-Bus. This vector object is then passed to Bluetit and processed by the "Options and Command Manager" implemented with a dedicated class. The option manager is invoked by the D-Bus `"set_options"` method used by a client to set the configuration of a session, whereas Bluetit commands are parsed in its D-Bus main loop.

Figure 1.2 shows how Bluetit processes options and commands received from a client. The main Bluetit thread is dedicated to the D-Bus loop and runs until it receives a termination signal either from the `root` user or a controlling process, such as `systemd`.

When the client sends a vector of options to Bluetit or calls a D-Bus method, it is then processed and each command or option/value pair is evaluated, the associated task is run, and finally a reply is sent

```
dbus_main_loop
{
    wait_for_dbus_method_from_client

    if D-Bus method is "set_options"; then
        process_requested_commands_and_options
    else
        evaluate_and_process_other_dbus_methods

    reply_to_client
}
```

Figure 1.2: Bluetit main D-Bus loop

back to the client through D-Bus.

Figure 1.3 shows the option sequence needed to request a connection to the currently best AirVPN server located in Germany. As it can be seen, the `std::vector<std::string>` object is constructed in order to have seven elements and each containing an option, value or command. It is then up to Bluetit options and command manager to properly interpret and use the items contained in this vector object.

The structure is similar to the well-known `argc` and `argv` pair used to pass options to a C program `main()` function from the shell. Options and commands are sent to Bluetit without a preceding double dash, something needed when using a command line client, such as `Goldcrest`. As a matter of fact, the `Goldcrest` client receives the commands and options from `argc` and `argv` variables, it subsequently creates the `std::vector<std::string>` object with each value in `argv` (element number 0 excluded) by removing the double dash beforehand and finally adding the value to the vector.

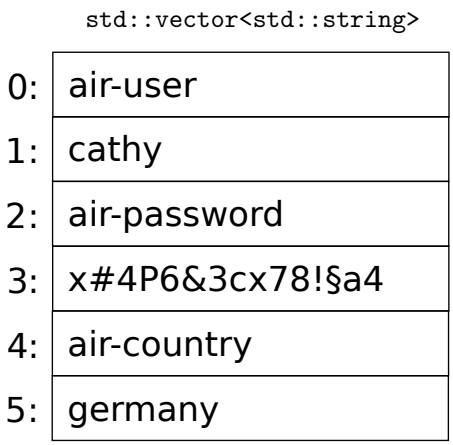

Figure 1.3: Bluetit option sequence for a connection to an AirVPN country

As it can be seen in figure 1.3 the sequence needed by Bluetit for starting a VPN connection, in this case, to the best server in Germany, is made of seven items. Each command or option (that is, all those beginning with `"air-"`) is represented without the preceding double dash. An option is always followed by a value, for example, `air-user` option is followed by `cathy` value), whereas a command (such as `ncp-disable`) does not have any associated value. It is up to Bluetit's options and command manager to properly manage this sequence and act as needed.

For the sake of completeness, the connection to the AirVPN session configured with the options shown in figure 1.3 is actually started by calling the D-Bus `"airvpn_start_connection"` method. For more information about AirVPN connection, refer to section 1.4.3 Starting a Connection to an AirVPN Server.

For a complete list and use of Bluetit's commands and options, read the AirVPN–SUITE user manual available at `https://airvpn.org/suite/readme` and part of the distribution package. In particular, the list of options and command, including examples for some specific cases, can be found in the `Goldcrest` section[5]. Each command or option can be used both in its long and short form. For example, option"air-server" (long form) has a corresponding short form option `"S"` (upper case letter "S") and they can be used interchangeably for the same purpose. This means that sending `"air-server"` or `"S"` to Bluetit has the effect of starting a connection process to an AirVPN server according to the other options and commands.

1.1.3 OpenVPN3 Service

The whole communication with VPN servers is realized with an encrypted and private tunnel, completely managed by OpenVPN3-AirVPN library, a direct fork from OpenVPN3[6] master branch and directly developed and maintained by AirVPN. Our fork has many advantages over the original branch, including – but not limited to – the fixing of some serious bugs preventing Linux from properly managing the connection and tunnel in some circumstances. Moreover, it offers new features, not available in the master branch, such as:

- `CHACHA20-POLY1305` cipher support for both control and data channels. A feature added in times when it was not available in the master branch and, in this regard, AirVPN has been the first one to add support for this cipher to OpenVPN3 as it was crucial for our Eddie Android application
- Cipher override to client configuration
- ncp disable override to client configuration
- `tcp-queue-limit` override to client configuration
- `ncp-disable` option in openvpn profile
- `data-ciphers` option in openvpn profile in order to comply to OpenVPN 2.5 negotiable data cipher specifications

[5] `https://airvpn.org/suite/readme/#goldcrest-client`

[6] OpenVPN and OpenVPN3 are Copyright © 2002-2021 OpenVPN Inc. `https://github.com/OpenVPN/openvpn3`

- Added support for DNS push ignore to Tunnel Builder

OpenVPN3-AirVPN fork, besides ensuring full compatibility with OpenVPN 2.x and the original OpenVPN3 master branch, has also been adapted and extended in order to suit the specific AirVPN needs and ensure fully interactivity and support with the AirVPN server infrastructure.

The OpenVPN3 service is completely managed by Bluetit and cannot be directly reached by the client as it is under the complete control of the daemon. OpenVPN3-AirVPN is part of Bluetit architecture and it is encapsulated at compilation time. In other words, it is not an external tool or library, it is an internal component. From an architectural point of view, the use of OpenVPN3 offers a safer and more secure model than actually calling an external OpenVPN binary. It should be said OpenVPN3 does not offer the wide range of options, functionality and features available, for example, in OpenVPN 2.5, however the fact it can be compiled, therefore encapsulated in a project as a class, undoubtedly offers a more secure and safer component, therefore limiting and preventing external events to actually exploiting inside a system. The choice of using OpenVPN3 over the binary counterpart of the 2.x series, certainly is more adequate in Bluetit as it is a component on its own, not to mention, it is run as a system daemon, therefore having `root` privileges which could become dangerous when improperly managed, such as in case of calling an external application.

Bluetit uses the OpenVPN3's `OpenVPN3Client` class by defining its own `VpnClient` class which is, in turn, created by inheriting this class as `public` as well as other specialized and internal classes.

Bluetit's `VpnClient` class is therefore the core object taking care of the VPN connection by using OpenVPN3. It of course extends the VPN functionality in order to suit its own needs as well as redefining and overriding all the virtual methods of the parent class.

1.1.4 AirVPN Manager

The AirVPN Manager is responsible for maintaining and ensuring communication with the AirVPN server infrastructure and services. It is actually made by a set of dedicated classes each responsible for a specific service or task, such as gathering information about the AirVPN user, servers and status.

This layer allows a client to fully interact with the AirVPN infrastructure by providing specific and relative options and commands. This component provides to the client full support with the interaction to the AirVPN infrastructure and in a completely transparent way by also processing and managing all the low level communication to the AirVPN system.

The client can interact with this component by sending Bluetit the corresponding options and commands by invoking the D-Bus `"set_options"` method as well as dedicated AirVPN D-Bus methods as explained in section 5.4 Public D-Bus Methods.

The AirVPN manager also takes care of the user login and logout procedures, as well as controlling the whole user session with the AirVPN servers. For example, in this specific case, the login and logout procedure is triggered by the "Options and Command Manager" when it receives the relevant options about a user login action. To clarify this, let's consider the example shown in figure 1.3. The first four items of the `std::vector<std::string>` object are directly passed to an AirVPN user object which actually starts the login request and session with the AirVPN infrastructure.

The AirVPN Manager is also in charge of keeping the list and performance of each server up-to-date in order to always provide reliable and latest information about the whole AirVPN's VPN infrastructure. It is also crucial for the *"quick connection"* procedure as it is strongly based on this information in determining the best server and according to the client request.

1.1.5 Network Lock and DNS Manager

This component is crucial to Bluetit as it provides and implements a *"best effort practice"* to prevent data leak, including traffic and DNS leaks. It is completely dependent on the hosting system and, in particular, with the available firewall and DNS manager running in the system.

It directly interacts with the system's firewall and DNS infrastructure by directly calling the corresponding tools in order to change and set the proper conditions and rules, therefore ensuring the *"best effort practice"* for data leak prevention. Bluetit currently supports the following firewall systems and infrastructure:

- `iptables` and `iptables-legacy`[7]
- `nftables`[8]

[7] `https://www.netfilter.org/projects/iptables/index.html`
[8] `https://www.netfilter.org/projects/nftables/index.html`

- `pf`[9]

`iptables`, `iptables-legacy` and `nftables` are the common *user-space* tools used by the system administrator to configure the IP packet filter rules of the Linux kernel firewall and implemented by the `Netfilter` modules. `pf` is the well-known "packet filter" firewall system developed for OpenBSD[10], then ported to other operating systems, such as FreeBSD[11] and Apple macOS™[12].

`Hummingbird` – the standalone AirVPN's OpenVPN3 client and part of the AirVPN–SUITE – uses most of the classes developed for this project and takes advantage of the "Network Lock and DNS Manager" therefore ensuring a *"best effort practice"* for Linux (by using `iptables`, `iptables-legacy` and `nftables`) as well as macOS (by using `pf`).

The "Network Lock and DNS Manager" can detect the system in which it is running as well as the firewall system to be used, save the case the client has set Bluetit to operate differently and with a different firewall system, including to turn it off. The "Network Lock and DNS Manager" actually uses a priority list in choosing the firewall system and tools. The higher priority is granted to `iptables-legacy`, then `iptables`, after that comes `nftables` and finally `pf`.

In case the client – by sending the option "networklock" with the D-Bus `"set_options"` method – or Bluetit configuration, sets the "Network Lock" mode to "auto", the first available firewall system and tools found according the internal priority list will be used. To summarize the priority list up again: `iptables-legacy` ⇒ `iptables` ⇒ `nftables` ⇒ `pf`. It is up to the "Network Lock and DNS Manager" to operate, set and manage the firewall system in a transparent way and according to AirVPN's firewall policy rules in order to ensure a *"best effort practice"* to prevent data leak.

This component is also responsible for the DNS management according to the *DNS push* information sent by the VPN server (either AirVPN's or generic) and therefore to properly set the hosting machine up. The Network Lock and DNS Manager can automatically detect and use the following DNS management tools and modes:

- `/etc/resolv.conf` file
- `systemd-resolved`[13]

The "Network Lock and DNS Manager" is also aware of Network Manager, in case it is running. It is important to note that any change to the DNS or firewall configuration done while Bluetit is connected to a VPN server strongly compromises the effect of the "Network Lock". For this reason it is strongly advised to not issue any DNS or firewall related command as long as Bluetit is connected.

In case of `resolv.conf`, the "Network Lock and DNS Manager" directly modifies the content of this file at the moment of VPN connection and restores it to its previous configuration at disconnection. In case the DNS management is under the control of `systemd-resolved`, the "Network Lock and DNS Manager" directly interacts with it by using the relative external tool and in the same fashion used for `resolv.conf`. This means the DNS is set according to the VPN connection needs and therefore restoring the original configuration at disconnection.

As for the macOS, the management of DNS is done internally in OpenVPN3 by directly calling the system's function controlling the DNS configuration – this is the specific case of `Hummingbird` – and therefore no external interaction with the system tools is required.

1.1.6 Logger

The logger is a quite straightforward component taking care of all Bluetit's logging needs. It is responsible both for sending messages to the system's log facility (that is, the `syslogd` daemon) as well as sending the relevant logging messages to the client.

All the logging messages generated by Bluetit are sent to the system log. Some of these messages are also sent to the client, depending on whether they are relevant or pertinent to the client scope.

Log messages are sent to the client through the D-Bus layer by invoking the `"log"` method. The client therefore needs to explicitly and specifically intercept this method and do the appropriate action in its D-Bus main loop, such as printing it in the terminal.

A Bluetit log message is processed by a D-Bus method and it is not considered an event as described in section 5.5 Bluetit Events. A Bluetit event – which is however implemented with a dedicated D-Bus method – signals the client the change of a condition or status – such as the disconnection from the VPN server – whereas a log message is a procedure requiring a specific and appropriate action.

[9] `https://www.openbsd.org/faq/pf/index.html`
[10] `https://www.openbsd.org`
[11] `https://www.freebsd.org`
[12] `https://www.apple.com/macos`
[13] `https://www.freedesktop.org/software/systemd/man/systemd-resolved.service.html`

Figure 1.4 shows an example on how a Bluetit log message should be processed by the client and by using the `DBusConnector` class[14]. A log message is *intercepted* by the client by processing the D-Bus `"log"` method and taking appropriate action, such as displaying the message in the terminal or by sending it to the client's log file.

1.2 Bluetit Sessions

Bluetit's distribution model is session dependent. This specifically means each task or action required to Bluetit corresponds to the creation of a new session. Moreover, this also means Bluetit usually serves one session at a time, although there are cases in which concurrent sessions can exist, such as in case of a VPN connection going on and the subsequent request of the connection statistics. The client must of course have granted a D-Bus connection with Bluetit D-Bus name. The D-Bus connection mode to Bluetit is always exclusive, that is there can be only one active client connected to the daemon. In case another client is going to start a D-Bus connection to Bluetit, D-Bus will not allow it because it is intentionally set to accept just one connection at a time. This mode is defined as "primary owner" in the D-Bus terminology and both Bluetit and clients connect to the D-Bus name by explicitly requesting to become the "primary owner".

Before starting a new session – that is, before invoking a Bluetit D-Bus method – the client must have successfully been connected to Bluetit's D-Bus name. As long as a client is connected to Bluetit via its D-Bus interface, no other client will be allowed to interact with the daemon. While the client is connected to the D-Bus, therefore is granted to be the "primary owner", it can start sessions by requesting any Bluetit valid D-Bus method, however depending on Bluetit configuration and settings.

Bluetit provides for three distinct session types:

- Internal session
- Synchronous client session
- Concurrent client session

[14] For more information, refer to chapter 4 The DBusConnector Class

```
#include <iostream>
#include <dbusconnector.hpp>

DBusConnector *dbusConnector;
DBusMessage *dbusMessage;
char *s;

....

while(dbusConnector->readWriteDispatch())
{
    while((dbusMessage = dbusConnector->popMessage()) != NULL)
    {
        if(dbusConnector->isMethod(dbusMessage, "log"))
        {
            if(dbusConnector->getArgs(dbusMessage, DBUS_TYPE_STRING, &s,
            DBUS_TYPE_INVALID))
                std::cout << dbusConnector->stringToLocale(s) << std::endl;
            else
                std::cerr << "ERROR: Cannot retrieve dbus log message" <<
                std::endl;
        }

        dbusConnector->unreferenceMessage(dbusMessage);
    }
}
```

Figure 1.4: Example of processing a Bluetit log message by using the `DBusConnector` class

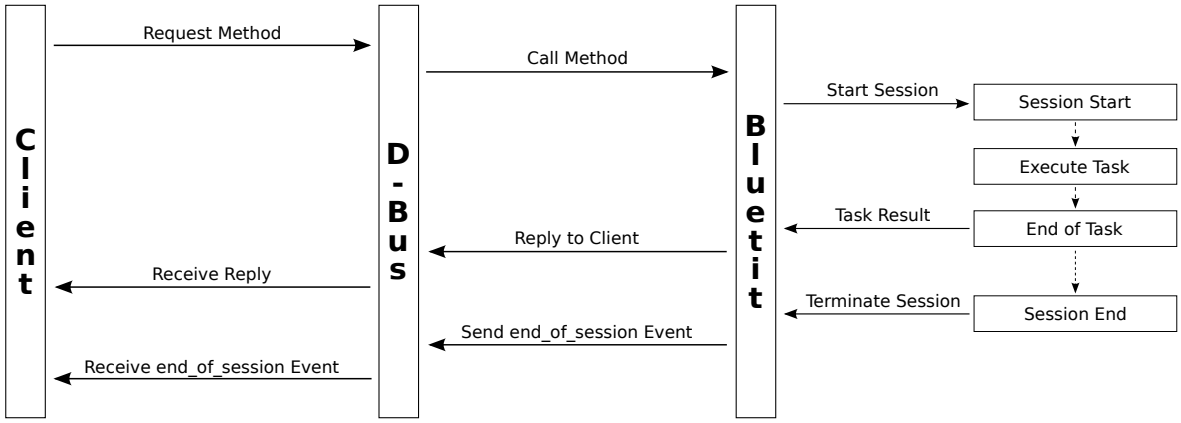

Figure 1.5: Bluetit client session diagram

Figure 1.5 shows the diagram about a session started by the client. It all starts with the client request sent to the D-Bus which subsequently dispatches it to Bluetit. From a client point of view, the operations needed to successfully start and complete a Bluetit session can be summarized as follows:

1. Request a Bluetit D-Bus method
2. Wait for a reply from D-Bus (if any)
3. Wait for `event_end_of_session`

Point 2 can be optionally skipped according to the way a D-Bus method is called. In case the client is developed by using the `DBusConnector` class, it depends on whether the method is called by using `callMethod()` or `callMethodWithReply()`, that is whether the D-Bus method returns a reply or not. In case the called method does not return any reply, this point can be skipped. As soon as Bluetit sends the "end of session" event, it is therefore ready to accept a new session from the client.

In general terms, a session is always started as a consequence of the options and commands sent to Bluetit by using the D-Bus `set_options` method as explained in section 1.1.2 Options and Command Manager, save the case of internal sessions which are always started by properly configuring Bluetit startup options in `bluetit.rc` file.

Finally, the start of a new session implicitly resets any option previously sent to Bluetit with the D-Bus `set_options` method. To be more precise, all Bluetit settings are always reset to their default value – the values set in `bluetit.rc` file are always considered as default – whenever a session is terminated. All the settings sent to Bluetit must always be considered *session settings* and are valid and effective for the currently associated session only.

1.2.1 Internal Session

An internal session simply is a session started by Bluetit, such as starting a connection to a VPN server at boot time. An internal session can be started by the `root` user only, for example by setting a connection at boot time by properly configuring the `bluetit.rc` file.

The client is never allowed to start an internal session, however it can control it, including termination, in case it is a session which can be controlled from the client.

For example, in the specific case of an internal session started with the VPN connection at boot, the client can eventually pause, resume and stop it. Every time a client controls the status of an internal session, it is actually starting a new client session.

1.2.2 Synchronous Client Session

A synchronous client session strictly follows the sequence shown in figure 1.5. Every session started by the client belongs to this type and, as a matter of fact, it is mandatory to wait for `event_end_of_session` before starting a new one.

This session type can also be used to control both the internal and concurrent sessions, such as pausing, resuming and terminating a VPN connection.

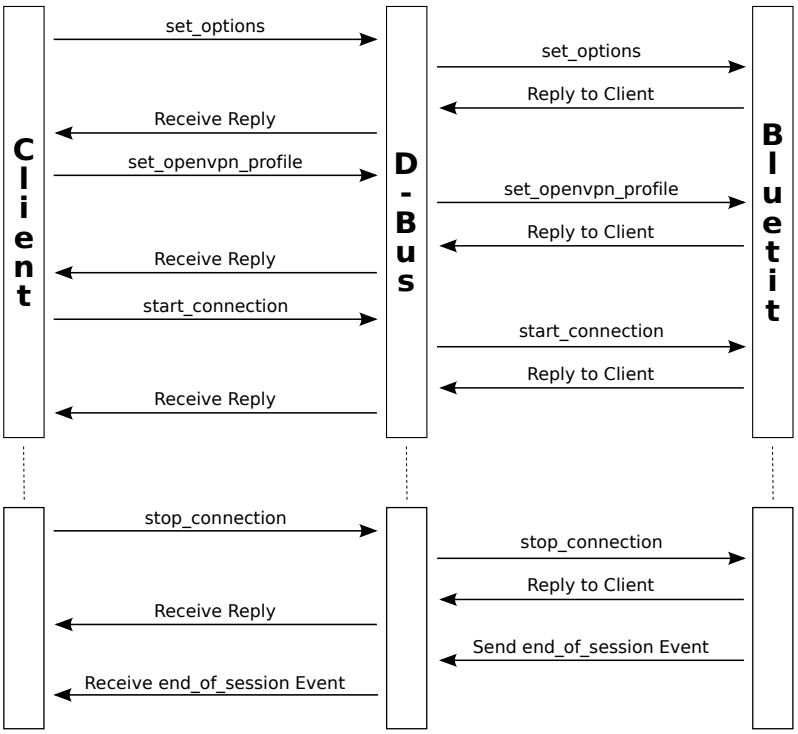

Figure 1.6: Starting an OpenVPN session

1.2.3 Concurrent Client Session

This particular type of session is always executed in Bluetit by starting a dedicated thread, therefore allowing the client to start new synchronous sessions. At the moment of writing this document, the only concurrent session type is associated to VPN connections, both to AirVPN and OpenVPN servers.

When a client requests a concurrent session, Bluetit returns the reply message (if any) and then waits for new incoming requests. Soon after a concurrent session has terminated – for example, when the VPN is disconnected – Bluetit sends to the client the associated "`event_end_of_session`" event. In the case of VPN disconnection, the `event_end_of_session` is preceded by the "disconnection" event. For more information about Bluetit events, refer to section 5.5 Bluetit Events.

1.3 Starting a Generic Session

A generic session is a task requested to Bluetit and not involving any of the AirVPN services. It is generally used for starting a connection to a generic OpenVPN server, therefore totally unrelated to the AirVPN universe. This also is the type of session a client would use to start the connection to a generic OpenVPN server, for example.

In general terms, a generic session is any task requested to Bluetit by sending a non "AirVPN" option, that is not starting with the `"air-"` prefix.

1.3.1 Starting an OpenVPN Connection

Starting a connection to an OpenVPN server is considered a special case of a generic session. Besides optionally using the D-Bus `set_options` method, the client also needs to send a valid OpenVPN profile – also known as "configuration file" – to Bluetit. The OpenVPN profile **must** be sent to Bluetit before actually starting the connection session by using the D-Bus `start_connection` method. The OpenVPN profile is sent to Bluetit by calling the D-Bus `set_openvpn_profile` method.

Figure 1.6 shows the diagram about a generic OpenVPN connection to a server. The process, from a client point of view, is made of three steps to establish a connection to a generic OpenVPN server, it then follows a *wait time* – that is, as long as the connection is needed and held – after which a "stop connection" request to Bluetit is sent and finally acknowledging `event_end_of_session`.

The first two steps of the procedure needs to be further discussed. The initial sequence of the process is about setting options and a valid OpenVPN profile. The call to D-Bus `set_options` method can be considered optional in case the client needs to override or set specific options to be used for the connection. An OpenVPN profile also defines options which are used by Bluetit for the connection process. For this reason, in case the OpenVPN profile contains all the required options for a connection, the call to D-Bus `set_options` method is not required.

All the options contained in the OpenVPN profile are actually changing Bluetit session options and, in some regards, it is equivalent to calling D-Bus `set_options` method and by providing those options.

Furthermore, the options set by calling D-Bus `set_options` method always have a higher priority over an OpenVPN profile and, in this specific case, the options set with the D-Bus method are actually an override to the corresponding profile options.

Of course, neither D-Bus `set_options` method or OpenVPN profile can override Bluetit's configuration options which are, in turn, unchangeable and always take the highest and ultimate priority. Bluetit's settings are defined in its run control file `bluetit.rc` and set by the system administrator `root` user.

The OpenVPN session is normally terminated by the client when it requests the disconnection from the server. This is easily done by calling the D-Bus `stop_connection` method, waiting for the associated reply and finally acknowledging `event_end_of_session`.

A connection can also be terminated by Bluetit as a consequence of some network conditions or errors, including the case the VPN server is terminating the connection. The client must be aware of this condition and properly manage it, for example, by processing `event_disconnected`. For more information about Bluetit events, refer to section 5.5.

1.4 Starting an AirVPN Session

An AirVPN session is not so different from a generic one in terms of procedure. The differences from the generic Bluetit session can be summarized like this:

- Use of one or more `"air-*"` options[15]
- Login to AirVPN infrastructure by providing user name and password
- Use of AirVPN related D-Bus methods, when needed and according to the required service

Every time a client sends to Bluetit and sets one or more `"air-*"` options, the session is always referred as "AirVPN session". This type of session does not however differ from any Bluetit session and follows the cycle of the diagram shown in figure 1.5.

1.4.1 AirVPN Options and Settings

Whenever a client sets any of the `"air-*"` options, the session is always referred as "AirVPN session". In this session type, non `"air-*"` options can be used as well and affect or change the overall configuration accordingly.

A special mention should be said about the `Goldcrest` option `"air-connect"`. This is the only option defined for this client to have no effect on Bluetit as it simply is an internal `Goldcrest` triggering option in order to let it call the D-Bus `airvpn_start_connection` method when specified in the command line.

For more information on how setting and configure a session, refer to section 1.1.2 Options and Command Manager.

1.4.2 Login to AirVPN Infrastructure

An AirVPN session requires a login procedure to the AirVPN infrastructure. The login procedure is simply done by sending to Bluetit both `air-user` and `air-password` options, properly assigned to valid user information. These data are passed to Bluetit via D-Bus and in *plain* format, in other words, both user name and password are not encrypted before sending them to the daemon.

Figure 1.3 shows a set of options used for an AirVPN session about a connection to the current best server in Germany. As long as the login procedure is concerned, only items from 0 to 3 are essential for the process.

[15] For a complete list of Bluetit and AirVPN related options, refer to AirVPN–SUITE User Manual available at `https://airvpn.org/suite/readme/#goldcrest-client`

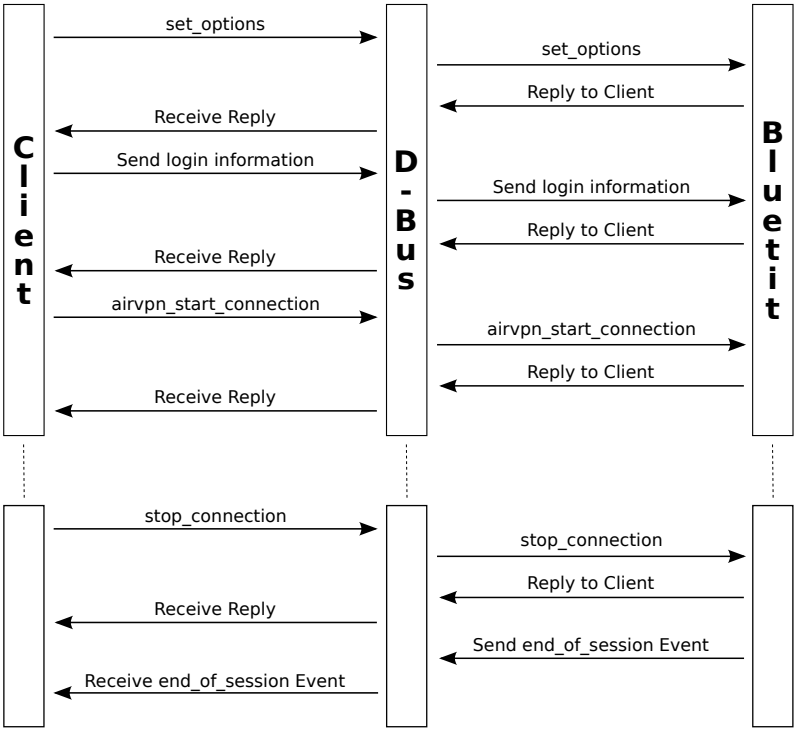

Figure 1.7: Starting an AirVPN session

These four items in fact define and set the essential information needed for completing a valid login procedure. The actual login to the AirVPN infrastructure is performed at the moment of the relative command execution. In case of the session configuration specified with the options shown in figure 1.3, the login to the AirVPN infrastructure is performed when the client calls the D-Bus `airvpn_start_connection` method. For the sake of completeness, the login procedure is performed before the actual start of the connection process.

1.4.3 Starting a Connection to an AirVPN Server

Bluetit provides for three distinct connection modes to the AirVPN infrastructure. The connection mode is determined by the options set for the AirVPN session and it must always be started with the D-Bus `airvpn_start_connection` method.

The currently provided connection modes to AirVPN servers are:

- Quick connection
- Server connection
- Country connection

Information about proper configuration and use for each connection mode is discussed in the dedicated sections below. Figure 1.7 shows the diagram for an AirVPN session. The procedure is the same for all modes which differ, one from each other, only for the options set for the session.

The notable differences from a generic session – shown in figure 1.6 – are represented by the setting of login information and the call to D-Bus `airvpn_start_connection` method. The disconnection procedure is just the same and involves a call to the D-Bus `stop_connection` method and finally acknowledging `event_end_of_session`.

The diagram shows the setting of options and login information as two separate tasks, both requiring a call to D-Bus `set_options` method. In case the client needs to set specific options besides providing information for AirVPN login, the `std::vector<std::string>` object can be filled with all the needed options and call the D-Bus `set_options` method just once.

The reason why the diagram in figure 1.7 shows these operations as separate tasks is because – as explained in the below sections – not all the AirVPN connection modes require the setting of specific options whereas all of them require the login information to be provided by the client.

1.4.3.1 Starting a Quick Connection

AirVPN quick connection provide a simple, automatic and straight method to connect to the current best server of the infrastructure and according to the geographical location of the client. Quick connection does not require any specific settings and, as a matter of fact, no `"air-*"` option must be used, except for login related options.

The client can however use any non `"air-*"` option in order to set specific connection parameters, such as port and protocol.

A quick connection requires the following steps and data:

- Set `air-user` option to the actual AirVPN user name
- Set `air-password` option to the AirVPN user password
- Set the optional non `"air-*"` options to their associated values
- Call D-Bus `set_options` method
- Call D-Bus `airvpn_start_connection` method

The quick connection procedure is based on an internal Bluetit algorithm taking into account the geographical location of the machine in which the daemon is running and, more specifically, the location of the Internet service providing the physical connection to the network.

In case the `root` user does not explicitly configure a geographical location in the `bluetit.rc` file, Bluetit will attempt to automatically determine the current location by inquiring AirVPN's `https://ipleak.net`. You should however be warned the automatic detection of the location – despite the target service is directly managed and owned by AirVPN – implies an external network access and Bluetit performs this action at boot time, that is, before any actual VPN connection.

The following scenarios should be considered:

- Bluetit persistent Network Lock is enabled
- External network access outside the encrypted tunnel

The persistent Netwotk Lock is enabled by Bluetit at boot time, therefore preventing any external network access and this includes Bluetit as well. In this specific case, it will be impossible for Bluetit to inquire AirVPN's `https://ipleak.net` and, as a consequence, the geographical location of the machine will not be determined. In this specific case, the location will be undetermined and the quick connection algorithm will consider the whole AirVPN infrastructure, therefore connecting to the current best AirVPN server in the world, possibly resulting in an inefficient VPN connection.

The second scenario should be considered carefully. The determination of the location is done at Bluetit boot time and in a phase in which there is no VPN connection active, including the "boot connection" optionally set in `bluetit.rc` file. This means reaching `https://ipleak.net` requires a plain DNS access and then an encrypted transaction – done with secure `HTTP` – with the AirVPN's website. The whole transaction is however done "as plain" and outside the encrypted tunnel.

Because of the above reasons, it is always suggested to manually set the geographical location of the machine in `bluetit.rc` file.

1.4.3.2 Starting a Connection to a Specific Server

Connecting to a specific AirVPN server actually consists in providing the very same setup used for quick connection and further providing both the `air-server` option and server name to which the client wants to connect to. A connection to a specific AirVPN server requires the following steps and data:

- Set `air-user` option to the actual AirVPN user name
- Set `air-password` option to the AirVPN user password
- Set any required options to their associated values
- Set `air-server` option to the name of the AirVPN server for which is requested the connection
- Call D-Bus `set_options` method
- Call D-Bus `airvpn_start_connection` method

As for AirVPN server names, they can be provided in any letter case mode, in other words, the evaluation of the server name is always case insensitive. For a complete and up-to-date list of available AirVPN servers, refer to `https://airvpn.org/status` or, alternatively, use the `Goldcrest` client with the below options:

```
$ goldcrest --air-list --air-server all
```

1.4.3.3 Starting a Connection to a Specific Country

Connecting to a specific AirVPN country means to request a connection to the best and more efficient server in a country at that specific time. The choice of the best server for each country is determined from AirVPN and it is the result of a periodic task in which all the information about servers are gathered, processed and then sorted in order to provide the user the most reliable information about the best eserver for each country or continent and, of course, the whole AirVPN infrastructure which is referred as "earth".

Starting an AirVPN connection to one of the available countries is not much different from connecting to a specific server. It basically consists of using the `air-country` option in place of `air-server`, as explained in the previous section.

A connection to a specific AirVPN country requires the following steps and data:

- Set `air-user` option to the actual AirVPN user name
- Set `air-password` option to the AirVPN user password
- Set any required options to their associated values
- Set `air-country` option to the name of the AirVPN country for which is requested the connection
- Call D-Bus `set_options` method
- Call D-Bus `airvpn_start_connection` method

As for AirVPN country names, they can be provided both with their actual name (for example, "Spain") and their corresponding ISO 3166 Alpha-2 code (for example, "ES"). The value can be expressed in any letter case mode, therefore the evaluation of the country or continent name, as well as the ISO code, is always case insensitive. For a complete and up-to-date list of available AirVPN countries, refer to `https://airvpn.org/status` or, alternatively, use the `Goldcrest` client with the below options:

```
$ goldcrest --air-list --air-country all
```

As for continent names, these are the currently valid and accepted values for `air-country` option for Bluetit version 1.1.0:

- `earth` (the whole AirVPN server infrastructure)
- `europe`
- `asia`
- `america` (the whole American continent, including south, central and north)

1.4.4 AirVPN Logout

The logout procedure is automatically done by Bluetit and, therefore, there is no need for the client to call any logout related method or command.

The procedure is always and automatically performed by Bluetit at the end of each session and it is completed just before sending event_end_of_session to the client. In other word, whenever the client receives this event, it also means the associated user has been successfully logged out from the AirVPN infrastructure and Bluetit is therefore ready to accept a new session and start the associated task.

Chapter 2

The DBusConnectorException Class

The `DBusConnector` class can throw exceptions in case of specific and critical conditions. For this purpose, the `DBusConnectorException` class is used whenever a critical error or condition arises and therefore causing the `DBusConnector` object to not complete its task.

The `DBusConnectorException` is a standard C++ class derived from `std::exception` class and inherits all its members as `public`.

2.1 Public Methods

2.1.1 DBusConnectorException()

Class constructor

```
DBusConnectorException(const std::string &errorMessage)
DBusConnectorException(const char *errorMessage)
```

Constructs a `DBusConnectorException` object.

Arguments:

errorMessage: Exception error message.

Example 2.1: Throwing a `DBusConnectorException`

```
#include <dbusconnector.hpp>

throw(DBusConnectorException("Failed to connect to D-Bus"));
```

2.1.2 ~DBusConnectorException()

Class destructor

```
~DBusConnectorException()
```

Destroys a `DBusConnectorException` object.

2.1.3 what()

Exception description

const char `*what()`

Returns the exception message associated to the `DBusConnectorException` object.

Return:

const char *: A pointer to a C-string with content related to the exception. This is guaranteed to be valid at least until the `DBusConnectorException` object from which it is obtained is destroyed or until a non-const member function of the `DBusConnectorException` object is called.

Chapter 3

The DBusResponse Class

The `DBusResponse` class is the main and preferred way used by Bluetit to provide a meaningful response to the client at the end of a request, as explained in section 5.4 Public D-Bus Methods.

The class allows the construction of datasets, including complex data sets made of unrelated and non homogeneous data types and contexts. A `DBusResponse` object can be seen as response message and a dataset represented by a vector of items where each of them can be, in turn, made of a single element or a set of information. A `DBusResponse` object always has a response message associated to it, whereas the dataset vector is optional and according to the use and data represented by the object.

The response message is a standard C++ string which can be conveniently used to return to the caller the exit status or error message as well as a "tag" identifying the dataset type. For specific `DBusResponse` datasets used by Bluetit, read section 5.6 Response Dataset Identities.

Each element of the dataset item is a named entity to which its relative and exclusive data value is associated. Figure 3.1 shows the typical structure of a `DBusResponse` object.

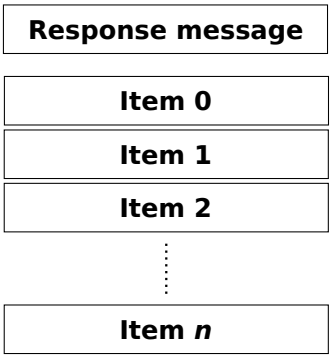

Figure 3.1: The structure of a `DbusResponse` object

Figure 3.2: Example of a `DbusResponse` object

Each item of the `DBusResponse` dataset is a group of data on its own and represents a set of information which could also be unrelated to all the other items of the object. Figure 3.2 shows a basic example of the structure of a possible `DBusResponse` object.

Each item is a set of named data (elements) and basically representing an `Item` data type of the `DbusResponse` class. Each Item corresponds to a `std::map` object, therefore a sorted associative container containing key–value pairs with unique keys. Each item is in fact defined as ***std::map<std::string, std::string>*** where the first element is the name (key) associated to the data contained in the second element.

This means a `DBusResponse Item` can be treated, used and compared as a typical and standard `std::map` object, including all the standard C++ functions available for processing this kind of object, such as standard iterators. The `DBusResponse` class however provides for iterators both for the whole dataset and the single items making it.

The class also provides for public methods to be used for creating and manipulating the internal dataset and every single element, therefore making its data types to be completely managed internally without the need of using the corresponding standard C++ methods. In order to ensure a better integrity as well as to provide a complete self-managed class, the public iterator methods are actually wrappers to the corresponding standard C++ counterparts.

Moreover, the `DBusResponse` class provides for methods allowing the marshaling of the internal dataset, therefore allowing a `DBusResponse` object to be serialized, made persistent or transmitted over the Internet or any IPC mechanism, both local and remote, such as D-Bus.

3.1 Public Types

3.1.1 Item

`DBusResponse` item

```
typedef std::map<std::string, std::string> Item
```

Defines a single item of the dataset associated to the response returned by the `DBusConnector` object at the end of a request. It uses a standard C++ map where the first element is the name of each item and second one is the associated value.

3.1.2 ItemIterator

Standard C++ `Item` iterator

```
typedef Item::iterator ItemIterator
```

Defines the iterator for a single `Item` in the `DBusResponse Item` dataset.

3.1.2.1 begin()

Standard C++ `ItemIterator` iterator

```
ItemIterator begin(Item item)
```

Defines the begin iterator for the specified `item` in the `DBusResponse Item` dataset.

Arguments:

item: The dataset item for which the `ItemIterator` is requested for.

Return:

ItemIterator: Iterator pointer to the beginning of `ItemIterator`, that is the first element in the `item` object.

Example 3.1: Iterating a `DBusResponse::Item` object

```
#include <iostream>
#include <dbusconnector.hpp>

DBusResponse::Item item;

for(DBusResponse::ItemIterator it = item.begin(); it != item.end(); it++)
{
    std::cout << "Name: " << it->first << " - Value: " << it->second <<
    std::endl;
}
```

3.1.2.2 end()

Standard C++ `ItemIterator` iterator

```
ItemIterator end(Item item)
```

Defines the end iterator for the specified `item` in the `DBusResponse Item` dataset.

Arguments:

item: the dataset item for which the `ItemIterator` is requested for.

Return:

ItemIterator: Iterator pointer to the end of `ItemIterator`, that is the last element in the `item` object.

Example 3.2: Using a `DBusResponse::ItemIterator`

```
#include <iostream>
#include <dbusconnector.hpp>

DBusResponse::Item item;

for(DBusResponse::ItemIterator it = item.begin(); it != item.end(); it++)
{
    std::cout << "Name: " << it->first << " - Value: " << it->second <<
    std::endl;
}
```

3.1.3 Iterator

Standard C++ `Item` iterator

```
typedef std::vector<Item>::iterator Iterator
```

Defines the iterator for the whole `DBusResponse Item` dataset.

3.1.3.1 begin()

Standard C++ `Item` iterator

```
Iterator begin()
```

Defines the begin iterator for the `DBusResponse Item` dataset.

Return:

Iterator: Iterator pointer to the beginning of `Iterator`, that is the first `Item` object in the `DBusResponse Item` dataset.

Example 3.3: Iterating the whole set of items in a `DBusResponse` object

```
#include <iostream>
#include <dbusconnector.hpp>

DBusResponse::Item item;

....

for(DBusResponse::Iterator it = dbusResponse->begin(); it !=
dbusResponse->end(); it++)
{
```

```
    item = *it;

    std::cout << "Name: " << it.first << " - Value: " << it.second <<
    std::endl;
}
```

3.1.3.2 end()

Standard C++ `Item` iterator

Iterator `end()`

Defines the end iterator for the `DBusResponse Item` dataset.

Return:

Iterator: Iterator pointer to the end of `Iterator`, that is the last `Item` object in the `DBusResponse Item` dataset.

Example 3.4: Using a `DBusResponse::Iterator`

```
#include <iostream>
#include <dbusconnector.hpp>

DBusResponse *dbusResponse = new DBusResponse();
DBusResponse::Item item;

....

for(DBusResponse::Iterator it = dbusResponse->begin(); it !=
dbusResponse->end(); it++)
{
    item = *it;

    std::cout << "Name: " << it.first << " - Value: " << it.second <<
    std::endl;
}
```

3.2 Public Methods

3.2.1 DBusResponse()

Class constructor

`DBusResponse()`

Constructs an empty `DBusResponse` object.

Example 3.5: Creating an empty `DBusResponse` object

```
#include <dbusconnector.hpp>

DBusResponse dbusResponse = new DBusResponse();
```

3.2.2 ~DBusResponse()

Class destructor

```
~DBusResponse()
```

Destroys a `DBusResponse` object.

Example 3.6: Destroying a `DBusResponse` object

```
#include <dbusconnector.hpp>

DBusResponse *dbusResponse;

DBusResponse dbusResponse = new DBusResponse();

...

delete dbusResponse;
```

3.2.3 clear()

Clears all the data contained in a `DBusResponse` object

```
void clear()
```

Clears and destroys all the items and any associated data currently contained in a `DBusResponse` object.

Example 3.7: Deleting all items in a `DBusResponse` object

```
#include <dbusconnector.hpp>

DBusResponse *dbusResponse;

// Remove all data in dbusResponse object

dbusResponse->clear();
```

3.2.4 setResponse()

Sets the response message

```
void setResponse(std::string value)
```

Sets the response message of a `DBusResponse` object and representing the exit status/result of the operation associated to the `DBusResponse` object. It can be any `std::string` representing a value globally accepted within the project.

Arguments:

value: A standard C++ string representing the exit/result status.

Example 3.8: Setting the response message of a `DBusResponse` object

```
#include <dbusconnector.hpp>
```

```
DBusResponse *dbusResponse;
bool success;

....

if(success == true)
{
    // Operation ended successfully

    dbusResponse->setResponse("OK");
}
else
{
    // Operation failed

    dbusResponse->setResponse("ERROR");
}
```

3.2.5 getResponse()

Gets the response message

std::string `getResponse()`

Gets the response message of a `DBusResponse` object and representing the exit status/result of the operation associated to the `DBusResponse` object.

Return:

std::string: Exit/result message associated to the response.

Example 3.9: Getting the response message associated to a `DBusResponse` object

```
#include <dbusconnector.hpp>
#include <iostream>

DBusResponse *dbusResponse;
std::string message;

....

message = dbusResponse->getResponse();

if(message == "OK")
    std::cout << "Operation ended successfully" << std::endl;
else
    std::cout << "Operation failed" << std::endl;
```

3.2.6 add()

Adds an `Item` to the dataset

void `add(`***Item*** `item)`

Adds (appends) an `Item` object to the dataset of the associated `DBusResponse` object.

Arguments:

Item: Item object to be added to the dataset.

Example 3.10: Adding an item to a `DBusResponse` dataset

```
#include <dbusconnector.hpp>

DBusResponse *dbusResponse;
DBusResponse::Item item;

....

dbusResponse->add(item);
```

3.2.7 addToItem()

Adds a new element to an `Item` of the dataset

```
void addToItem(Item &item, std::string key, std::string value)
```

Adds a new element (key–value pair) to an `Item` object.

Arguments:

item: Item object to which the element is to be added.
key: Key name of the element.
value: Value associated to the key.

Example 3.11: Addding a new element to an item for a `DBusResponse` dataset

```
#include <dbusconnector.hpp>

DBusResponse *dbusResponse;
DBusResponse::Item item;

....

dbusResponse->addToItem(item, "Server", "Diadema");
dbusResponse->addToItem(item, "Country", "Belgium");

dbusResponse->add(item);
```

3.2.8 getItem()

Gets an `Item` of the dataset

```
Item getItem(int row)
```

Gets an `Item` object of the dataset.

Arguments:

row: Number (index) of the element to be retrieved, starting from 0.

Return:

Item: Item associated to the "row" entry in the dataset. It returns an empty `Item` in case `row` is out of range.

Example 3.12: Get a specific item from a `DBusResponse` dataset

```
#include <dbusconnector.hpp>

DBusResponse *dbusResponse;
DBusResponse::Item item;

....

item = dbusResponse->getItem(1);
```

3.2.9 rows()

Number of elements in dataset

```
int rows()
```

Returns the number of elements (rows) in the dataset.

Return:

int: Number of items currently contained in `DBusResponse` object.

Example 3.13: Get the item count in a `DBusResponse` dataset

```
#include <dbusconnector.hpp>

DBusResponse *dbusResponse;
int rows;

....

rows = dbusResponse->rows();
```

3.2.10 items()

Number of key-value pairs in item

```
int items(int row)
```

Returns the number of key–value pairs contained in the indexed item of the dataset.

Arguments:

int: Index (row) number of the `Item` of dataset.

Return:

int: Number of key-value pairs contained in the `Item` object.

Example 3.14: Get the count of elements in a `DBusResponse` item

```
#include <dbusconnector.hpp>
```

```
DBusResponse *dbusResponse;
Item item;
int rows;

....

item = dbusResponse->getItem(2);

rows = dbusResponse->items(item);
```

3.2.11 getItemValue()

Value of a named element

std::string `getItemValue(`***Item*** `item, `***std::string*** `key)`

Returns the value associated to the element named "key" in an `Item`.

Arguments:

item: `Item` object.
key: "key name" of the element of which retrieve value.

Return:

std::string: Value associated to the "key". It returns an empty string in case "key" does not exist in the `Item`.

Example 3.15: Get the value of a named element in a `DBusResponse` item

```
#include <dbusconnector.hpp>
#include <iostream>

DBusResponse *dbusResponse;
Item item;
std::string serverName;

....

item = dbusResponse->getItem(1);

serverName = dbusResponse->getItemValue(item, "Server");

if(serverName != "")
    std::cout << "Server name: " << serverName << std::endl;
else
    std::cout << "Server is undefined" << std::endl;
```

3.2.12 itemKey()

Key name of `ItemIterator`

std::string `itemKey(`***ItemIterator*** `it)`

Returns the key name associated to an `ItemIterator` object.

Arguments:

it: `ItemIterator` object.

Return:

std::string: Value of the `ItemIterator` object.

Example 3.16: Get the key name of a `DBusResponse::ItemIterator`

```
#include <iostream>
#include <dbusconnector.hpp>

DBusResponse::Item item = dbusResponse->getItem(3);

for(DBusResponse::ItemIterator it = item.begin(); it != item.end(); it++)
{
    std::cout << "Key: " << dbusResponse->itemKey(it) << std::endl;
}
```

3.2.13 itemValue()

Value of `ItemIterator`

std::string `itemValue(`***ItemIterator*** `it)`

Returns the value associated to an `ItemIterator` object.

Arguments:

it: `ItemIterator` object.

Return:

std::string: Key name of the `ItemIterator` object.

Example 3.17: Get the value of a `DBusResponse::ItemIterator`

```
#include <iostream>
#include <dbusconnector.hpp>

DBusResponse::Item item = dbusResponse->getItem(0);

for(DBusResponse::ItemIterator it = item.begin(); it != item.end(); it++)
{
    std::cout << dbusResponse->itemKey(it) " = " << dbusResponse->itemValue(it)
    << std::endl;
}
```

3.2.14 fromString()

Converts a marshaled string into a `DBusResponse` dataset

bool `fromString(`***std::string*** `str)`

Converts a marshaled string, previously created with `toString()`, into the `DBusResponse` dataset.

Arguments:

std::string: Marshaled string.

Return:

bool: `true` in case of successful conversion, `false` in case of error or malformed string.

Example 3.18: Populate a `DBusResponse` object from string

```
#include <dbusconnector.hpp>

// Populate the datatet with the marshaled string received from D-Bus

dbusResponse->fromString(str);
```

3.2.15 toString()

`DBusResponse` dataset marshaler

```
std::string toString()
```

Converts the `DBusResponse` dataset into a marshaled string suitable to be sent over D-Bus, stored or serialized into a persistent mean.

Return:

std::string: Marshaled string. It returns an empty string in case the `DBusResponse` dataset is empty.

Example 3.19: Convert a `DBusResponse` object into a string

```
#include <dbusconnector.hpp>

std::string dataset;

// Marshal the current dataset into a string

dataset = dbusResponse->toString();
```

Chapter 4

The DBusConnector Class

`DBusConnector` is the core class used by Bluetit for all the D-Bus activity. This class is however independent from the AirVPN–SUITE although it has specifically been designed and developed in order to meet and satisfy Bluetit inter-process communication needs.

It is completely based on standard D-Bus specifications[1] and it internally uses low-level D-Bus C API[2] as defined in version 1.13.

This ensures full compatibility with any D-Bus system or infrastructure, provided it fully complies to version 1.13 specifications. This does mean D-Bus communication with Bluetit daemon can be achieved with any programming language having a full support to D-Bus and implementing 1.13 specifications.

It should however be noted that a client application – regardless of the programming language and system used for the development – must obey to and follow Bluetit architecture requirements as well as complying to its D-Bus message format, objects and conventions.

Specifically, any client application needs to comply to Bluetit's Public D-Bus Methods[3] in order to *request* any service to Bluetit by invoking its public D-Bus methods and by providing arguments and values according to the specifications covered in section 5.4.

Likewise, the client application needs to be developed in order to properly process Bluetit D-Bus responses according to standard D-Bus `DBusMessage` structure and `DBusResponse`[4] class, this latter being covered in chapter 3.

The above considerations of course apply in case the client application is going to be developed in a language different from C++, otherwise the developer can use the D-Bus classes part of AirVPN–SUITE, specifically `DBusConnector`[5], `DBusResponse`[6] and `DBusConnectorException`[7] classes.

The `DBusConnector` class provides methods in order to ensure a complete and full D-Bus support and takes care of all the low level D-Bus operations needed for the inter-process communication to and from the client, including opening, managing and closing the connection with the D-Bus system daemon.

This class may throw exceptions in case of errors or critical conditions and, in such cases, the developer needs to properly *catch* and manage exceptions. All the exceptions thrown by the `DBusConnector` class are of type `DBusConnectorException`. Refer to each class method specification in order to see when and how exceptions may be thrown by this class.

4.1 Character Encoding and Messages

The `DBusConnector` class internally uses `UTF-8`[8] character encoding. All the messages, strings and character sequences provided to or received from the `DBusConnector` class are expected to be encoded in `UTF-8`.

[1] D-Bus is a message bus system, a simple way for applications to talk to one another. In addition to inter-process communication, D-Bus helps coordinate process lifecycle; it makes it simple and reliable to code a "single instance" application or daemon, and to launch applications and daemons on demand when their services are needed. For more information, refer to the official Website at `https://www.freedesktop.org/wiki/Software/dbus`

[2] `https://dbus.freedesktop.org/doc/api/html/index.html`

[3] refer to section 5.4 Public D-Bus Methods

[4] Refer to chapter 3 The DBusResponse Class

[5] Refer to chapter 4 The DBusConnector Class

[6] Refer to chapter 3 The DBusResponse Class

[7] Refer to chapter 2 The DBusConnectorException Class

[8] UTF-8, Unicode Transformation Format with 8 bits per code unit. `https://www.unicode.org/main.html`

In order to make sure all character and string data received by the `DBusConnector` can be processed by the class, the public methods receiving data from a client convert them into `UTF-8` before proceeding to further processing.

Likewise, all character and string data sent to a client are converted into `UTF-8` before actually sending the data to the D-Bus daemon. In case the client needs to convert `UTF-8` into its locale encoding, the `DBusConnector` provides for public conversion methods to be used for this purpose. All the internal character encoding conversions are done by using GNU's `libiconv`[9].

All the incoming messages – that is, arguments, options and values required by Bluetit public D-Bus methods – are expected to be represented by a standard C++ `std::vector<std::string>` object. Figure 4.1 shows an example on how arguments and values are expected to be passed to a Bluetit D-Bus method by using the `DBusConnector` class.

All the responses sent by `DBusConnector` to the caller may be both represented by a pointer to a standard `DBusMessage` structure and to a `DBusResponse` object and its associated exit (response) status. All D-Bus methods always return a pointer to a standard `DBusMessage` structure from which the caller can get access to the possible associated `DBusResponse` object, in case it is provided by the called D-Bus method.

Each `DBusMessage` may have its own `DBusResponse` object and, in both cases, they are owned by the caller. Specifically, each `DBusMessage` and the relative `DBusResponse` object are created by allocating a dedicated memory space in the system and the release of this allocated memory is on the sole responsibility of the caller. This means it is **mandatory for the client** to tell the `DBusConnector` object when the data contained in `DBusMessage` and `DBusResponse` have been processed and are not needed anymore.

The memory allocated for `DBusMessage` and `DBusResponse` objects is freed by calling the

[9] GNU `libiconv` and `libcharset` libraries and their header files are released under the `LGPL` (GNU Lesser General Public License). `https://www.gnu.org/software/libiconv`

```
#include <iostream>
#include <vector>
#include <dbusconnector.hpp>

DBusConnector *dbusConnector = nullptr;
DBusMessage *dbusReply = nullptr;
std::vector<std::string> dbusItems;

try
{
    dbusConnector = new DBusConnector("org.airvpn.dbus", "org.airvpn.server");
}
catch(DBusConnectorException &e)
{
    std::cerr << e.what() << std::endl;

    cleanup_and_exit();
}

dbusItems.clear();

dbusItems.push_back("air-connect");
dbusItems.push_back("air-country");
dbusItems.push_back("Netherlands");

dbusReply = dbusConnector->callMethodWithReply("org.airvpn.server",
"/org/airvpn/server", "set_options", dbusItems);

if(dbusReply == nullptr)
{
    // Error

    cleanup_and_exit();
}
```

Figure 4.1: Example of calling a Bluetit D-Bus method by using the `DBusConnector` class

```
#include <iostream>
#include <string>
#include <vector>
#include <dbusconnector.hpp>

DBusConnector *dbusConnector = nullptr;
DBusMessage *dbusReply = nullptr;
int server_status = -1;

dbusItems.clear();

dbusReply = dbusConnector->callMethodWithReply("org.airvpn.server",
"/org/airvpn/server", "bluetit_status", dbusItems);

if(dbusReply == nullptr)
{
    std::cout << "ERROR: Invalid reply" << std::endl;

    cleanup_and_exit();
}

server_status = dbusConnector->getInt(dbusReply);

dbusConnector->unreferenceMessage(dbusReply);
```

Figure 4.2: Example of getting the integer value associated to `DBusMessage` returned by a Bluetit D-Bus method and by using the `DBusConnector` class

`DBusConnector`'s methods `unreferenceMessage()` and `unreferenceResponse()` respectively.

Figure 4.2 shows an example on how to get the integer value associated to a `DBusMessage` and returned by a D-Bus method call, whereas figure 4.3 shows an example on how to get a `DBusResponse` object from a D-Bus method call and its associated exit status. Both examples are to be considered as a hypothetical continuation of the code shown in figure 4.1.

For more information on how to use Bluetit D-Bus methods, refer to section 5.4, whereas for information about `DBusResponse` class, please refer to chapter 3.

4.2 Public Methods

4.2.1 DBusConnector()

Class constructor

```
DBusConnector(std::string interface, std::string bus)
```

Constructs a `DBusConnector` object and connects to the specified D-Bus `interface` and `bus`.

Arguments:

interface: D-Bus interface name.
bus: D-Bus bus name.

Exceptions:

DBusConnectorException: Interface or bus is empty;
connection to D-Bus failed

Example 4.1: Contructing a `DBusConnector` object

```
#include <iostream>
```

```
#include <dbusconnector.hpp>

DBusConnector *dbusConnector = nullptr;

try
{
    dbusConnector = new DBusConnector("org.airvpn.dbus", "org.airvpn.server");
}
catch(DBusConnectorException &e)
{
    std::cerr << e.what() << std::endl;
}
```

4.2.2 ~DBusConnector()

Class destructor

```
~DBusConnector()
```

Destroys a `DBusConnector` object and closes the connection to the associated D-Bus `interface` and `bus`.

Example 4.2: Destroying a `DBusConnector` object

```
#include <iostream>
#include <dbusconnector.hpp>
```

```
#include <iostream>
#include <string>
#include <vector>
#include <dbusconnector.hpp>

DBusConnector *dbusConnector = nullptr;
DBusMessage *dbusReply = nullptr;
DBusResponse *dbusResponse = nullptr;

....

try
{
    dbusResponse = dbusConnector->getResponse(dbusReply);

    if(dbusResponse->getResponse() == "OK")
        std::cout << "Options successfully set" << std::endl;
    else
        std::cout << "Invalid options" << std::endl;
}
catch(DBusConnectorException &e)
{
    std::cout << "Cannot get response. D-Bus error: " << e.what() << std::endl;

    cleanup_and_exit();
}

dbusConnector->unreferenceResponse(dbusResponse);

dbusConnector->unreferenceMessage(dbusReply);
```

Figure 4.3: Example of getting a `DBusResponse` object from a `DBusMessage` returned by a `DBusConnector` object

```
DBusConnector *dbusConnector = nullptr;

try
{
    dbusConnector = new DBusConnector("org.airvpn.dbus", "org.airvpn.server");
}
catch(DBusConnectorException &e)
{
    std::cerr << e.what() << std::endl;
}

...

delete dbusConnector;
```

4.2.3 readWriteDispatch()

D-Bus message dispatcher

bool `readWriteDispatch(`***int*** `timeout_millis=50)`

In case there are D-Bus messages waiting to be dispatched, this method calls the D-Bus dispatcher once and then returns. The method waits for the `timeout_millis` before invoking the D-Bus low level dispatcher. This method is intended to be used in the application D-Bus main loop.

Arguments:

timeout_millis: Maximum wait time in milliseconds. It defaults to 50 milliseconds.

Return:

bool: `true` in case D-Bus disconnection message has not been processed, `false` otherwise

Example 4.3: Reading and dispatching D-Bus messages by using a `DBusConnector` object

```
#include <iostream>
#include <dbusconnector.hpp>

DBusConnector *dbusConnector;
DBusMessage *dbusMessage;

....

while(dbusConnector->readWriteDispatch())
{
    while((dbusMessage = dbusConnector->popMessage()) != NULL)
    {
        // Process your DBusMessage here

        dbusConnector->unreferenceMessage(dbusMessage);
    }
}
```

4.2.4 popMessage()

Gets the first D-Bus message from the queue

DBusMessage `*popMessage()`

Returns the first message from the incoming D-Bus message queue and then removes it from the queue.

Return:

DBusMessage *: Pointer to the retrieved message. The caller owns the reference to the returned message and must unreference it as soon as it is done processing the message by using `unreferenceMessage()`. In case the message queue is empty, it returns `NULL` (`nullptr`).

Example 4.4: Popping a D-Bus messages from the queue by using a `DBusConnector` object

```
#include <iostream>
#include <dbusconnector.hpp>

DBusConnector *dbusConnector;
DBusMessage *dbusMessage;

....

while(dbusConnector->readWriteDispatch())
{
    while((dbusMessage = dbusConnector->popMessage()) != NULL)
    {
        // Process your DBusMessage here

        dbusConnector->unreferenceMessage(dbusMessage);
    }
}
```

4.2.5 isMethod()

Checks whether a method call is valid

bool `isMethod(`***DBusMessage*** `*dbusMessage,` ***std::string*** `method)`

Checks whether the `DBusMessage` is a method call with the given `method` interface name.

Arguments:

dbusMessage *: Pointer to the D-Bus message structure which the `method` belongs to
method: Name of the method to be checked

Return:

bool: `true` in case the `dbusMessage` is a method call with the `method` interface name, `false` otherwise.

Example 4.5: Checking a D-Bus method by using a `DBusConnector` object

```
#include <dbusconnector.hpp>

DBusConnector *dbusConnector;
DBusMessage *dbusMessage;

....

while(dbusConnector->readWriteDispatch())
```

```
{
    while((dbusMessage = dbusConnector->popMessage()) != NULL)
    {
        if(dbusConnector->isMethod(dbusMessage, "start_connection"))
        {
            // process D-Bus method call
        }

        dbusConnector->unreferenceMessage(dbusMessage);
    }
}
```

4.2.6 callMethod()

Calls a D-Bus method

bool `callMethod(`***std::string*** `bus,` ***std::string*** `path,` ***std::string*** `method,`
std::vector<std::string> `item)`

Calls a D-Bus public method and returns immediately without waiting for D-Bus reply. This method is intended to call a D-Bus method whose interface does not provide for a reply.

Arguments:

bus: D-Bus bus name.
path: D-Bus path name.
method: Name of the public D-Bus method to be called.
item: Vector of arguments, options and data to be passed to the `method`

Return:

bool: `true` in case of successful operation, `false` in case the D-Bus connection is not available.

Exceptions:

DBusConnectorException: Method name not found or invalid;
Error in appending an `item` value;
D-Bus execution error

Example 4.6: Calling a D-Bus method by using a `DBusConnector` object

```
#include <iostream>
#include <string>
#include <vector>
#include <dbusconnector.hpp>

DBusConnector *dbusConnector = nullptr;
std::vector<std::string> dbusItems;
bool success = false;

....

dbusItems.clear();

dbusItems.push_back("Operation successful");

try
{
    success = dbusConnector->callMethod("org.airvpn.server",
    "/org/airvpn/server", "log", dbusItems);
```

```
}
catch(DBusConnectorException &e)
{
    // Error
}

if(success == false)
    std::cout << "ERROR: D-Bus connection is not available" << std::endl;
```

4.2.7 callMethodWithReply()

Calls a D-Bus method and returns the D-Bus reply

DBusMessage `*callMethodWithReply(`***std::string*** `bus,` ***std::string*** `path,`
std::string `method,` ***std::vector<std::string>*** `item)`

Calls a D-Bus public method, then waits for D-Bus reply and returns it.

Arguments:

bus: D-Bus bus name.
path: D-Bus path name.
method: Name of the public D-Bus method to be called.
item: Vector of arguments, options and data to be passed to the `method`

Return:

DBusMessage *: Pointer to the D-Bus message. The caller owns the reference to the returned message and must unreference it as soon as it is done processing the message by using `unreferenceMessage()`. `NULL` (`nullptr`) in case the D-Bus connection is not available.

Exceptions:

DBusConnectorException: Method name not found or invalid;
Error in appending an `item` value;
Null D-Bus reply;
D-Bus execution error

Example 4.7: Calling a D-Bus method returning a reply by using a `DBusConnector` object

```
#include <iostream>
#include <vector>
#include <dbusconnector.hpp>

DBusConnector *dbusConnector = nullptr;
DBusMessage *dbusReply = nullptr;
std::vector<std::string> dbusItems;

....

dbusItems.clear();

dbusItems.push_back("air-info");
dbusItems.push_back("air-server");
dbusItems.push_back("xuange");

try
{
    dbusReply = dbusConnector->callMethodWithReply("org.airvpn.server",
    "/org/airvpn/server", "set_options", dbusItems);
```

```
}
catch(DBusConnectorException &e)
{
    // Error
}

if(dbusReply == nullptr)
{
    // Error

    cleanup_and_exit();
}
```

4.2.8 replyToMessage()

Replies to a D-Bus message

bool `*replyToMessage(`***DBusMessage*** `*dbusMessage,` ***DBusResponse*** `dbusResponse)`
bool `*replyToMessage(`***DBusMessage*** `*dbusMessage,` ***std::vector<std::string>*** `item)`
bool `*replyToMessage(`***DBusMessage*** `*dbusMessage,` ***int*** `value)`

Replies to a pending D-Bus message in response to a `callMethodWithReply()` call.

Arguments:

dbusMessage *: `DBusMessage` pointer which the reply is related to.
dbusResponse: `DBusResponse` object associated to the reply and to be returned to the caller.
item: Vector of C++ strings associated to the reply and to be returned to the caller.
value: Integer value (number) associated to the reply and to be returned to the caller.

Return:

bool: `true` in case of successful reply, `false` in case of connection unavailable or invalid reply data.

Exceptions:

DBusConnectorException: Error in creating D-Bus reply;
Error in sending D-Bus reply;
Error in appending data to the return item

Example 4.8: Replying to a D-Bus message by using a `DBusConnector` object

```
#include <iostream>
#include <vector>
#include <dbusconnector.hpp>

DBusConnector *dbusConnector = nullptr;
DBusMessage *dbusReply = nullptr;
int status = 1;
bool success = false;

....

try
{
    success = dbusConnector->replyToMessage(dbusReply, status);
}
catch(DBusConnectorException &e)
{
    // Error
```

```
}

if(success == false)
{
    // Error
}
```

4.2.9 getArgs()

Gets DBusMessage argument va_list

bool getArgs(***DBusMessage*** *dbusMessage, ***int*** firstArgType, ...)

Gets and assigns DBusMessage data by using va_list

Arguments:

dbusMessage *: DBusMessage pointer which the reply is related to.

int: First argument of the va_list associated to the DBusMessage. It is expressed by a series of data pairs where the first one is a valid DBus type code and the second is the reference to the variable or object used to rececive the related data. The va_list must be terminated with the DBus type code DBUS_TYPE_INVALID.

Return:

bool: true in case of successful operation, false in case the D-Bus connection is not available or DBusMessage in NULL.

Example 4.9: Get the arguments associated to a D-Bus message by using a DBusConnector object

```
#include <iostream>
#include <dbusconnector.hpp>

DBusConnector *dbusConnector;
DBusMessage *dbusMessage;
char *s, int value;

....

if(dbusConnector->getArgs(dbusMessage, DBUS_TYPE_STRING, &s, DBUS_TYPE_INT,
&value, DBUS_TYPE_INVALID))
{
    // Error
}

std::cout << "First argument: " << s << " - Second argument: " << value <<
std::endl;
```

4.2.10 getVector()

Gets the string vector of a DBusMessage

std::vector<std::string> getVector(***DBusMessage*** *dbusMessage)

Returns a standard C++ vector of strings associated to a DBusMessage

Arguments:

dbusMessage *: DBusMessage pointer which the reply is related to.

Return:

std::vector<std::string>: Standard C++ vector of standard C++ strings associated to the `DBusMessage`

Example 4.10: Get the vector associated to a D-Bus message by using a `DBusConnector` object

```
#include <iostream>
#include <vector>
#include <string>
#include <dbusconnector.hpp>

DBusConnector *dbusConnector;
DBusMessage *dbusMessage;
std::vector<std::string> dbusItems;

....

dbusItems = dbusConnector->getVector(dbusMessage);

if(dbusItems.empty() == true)
    std::cout << "Data set is empty" << std::endl;
```

4.2.11 getInt()

Gets the integer value of a `DBusMessage`

```
int getInt(DBusMessage *dbusMessage)
```

Returns the integer value associated to a `DBusMessage`

Arguments:

dbusMessage *: `DBusMessage` pointer which the reply is related to.

Return:

int: Integer value associated to the `DBusMessage`

Example 4.11: Get the integer value associated to a D-Bus message by using a `DBusConnector` object

```
#include <iostream>
#include <vector>
#include <string>
#include <dbusconnector.hpp>

DBusConnector *dbusConnector;
DBusMessage *dbusMessage;
int value;

....

value = dbusConnector->getInt(dbusMessage);
```

4.2.12 getResponse()

Gets the `DBusResponse` object of a `DBusMessage`

```
DBusResponse *getResponseint(DBusMessage *dbusMessage)
```

Returns the pointer to the `DBusResponse` object associated to a `DBusMessage`. **Important notice**: the caller owns the returned `DBusResponse` object and it is on his or her sole responsibility to release (unreference) it when the object is not needed anymore. To unreference a `DBusResponse` object the caller must use the `unreferenceResponse()` method.

Arguments:

dbusMessage *: `DBusMessage` pointer which the reply is related to.

Return:

DBusResponse *: Pointer of the `DBusRespose` object associated to the `DBusMessage`

Example 4.12: Get the `DBusResponse` object associated to a D-Bus message by using a `DBusConnector` object

```
#include <dbusconnector.hpp>

DBusConnector *dbusConnector;
DBusMessage *dbusMessage;
DBusResponse *dbusResponse;

....

dbusResponse = dbusConnector->getResponse(dbusMessage);

.....

dbusConnector->unreferenceResponse(dbusResponse);
```

4.2.13 unreferenceResponse()

Unreferences a `DBusResponse` object

```
void unreferenceResponse(DBusResponse *dbusResponse)
```

Unreferences a `DBusResponse` object associated to a `DBusMessage` by releasing and freeing all the associated resources ad memory.

Arguments:

dbusResponse *: `DBusResponse` pointer to the object to be unreferenced.

Example 4.13: Unreferencing a `DBusResponse` object

```
#include <dbusconnector.hpp>

DBusConnector *dbusConnector;
DBusMessage *dbusMessage;
DBusResponse *dbusResponse;

....
```

```
dbusResponse = dbusConnector->getResponse(dbusMessage);

.....

dbusConnector->unreferenceResponse(dbusResponse);
```

4.2.14 unreferenceMessage()

Unreferences a `DBusMessage` structure

void `unreferenceMessage(`***DBusMessage*** `*dbusMessage)`

Unreferences a `DBusMessage` structure. Calling this method is mandatory as soon as the `DBusMessage` is not needed anymore.

Arguments:

dbusMessage *: `DBusMessage` pointer to the structure to be unreferenced.

Example 4.14: Unreferencing a D-Bus message by using `DBusConnector` object

```
#include <dbusconnector.hpp>

DBusConnector *dbusConnector;
DBusMessage *dbusMessage;

....

dbusMessage = dbusConnector->popMessage();

.....

dbusConnector->unreferenceMessage(dbusMessage);
```

4.2.15 stringToUTF8()

Converts a string into `UTF-8`

std::string `stringToUTF8(`***std::string*** `str)`

Converts a string into `UTF-8` by using `libiconv`'s `iconvString()` function

Arguments:

str: String to be converted

Return:

std::string: Converted string

Example 4.15: Converting a string into `UTF-8` encoding by using a `DBusConnector` object

```
#include <string>
#include <dbusconnector.hpp>

DBusConnector *dbusConnector;
```

```
std::string a, b;

....

b = dbusConnector->stringToUTF8(a);
```

4.2.16 stringToLocale()

Converts a string into the locale encoding

std::string `stringToLocale(`***std::string*** `str)`

Converts a string into the locale encoding of the system in which the method is called from and by using `libiconv`'s `iconvString()` function

Arguments:

str: String to be converted

Return:

std::string: Converted string

Example 4.16: Converting a string into locale encoding by using a `DBusConnector` object

```
#include <string>
#include <dbusconnector.hpp>

DBusConnector *dbusConnector;
std::string a, b;

....

b = dbusConnector->stringToLocale(a);
```

Chapter 5

Bluetit D-Bus Interface

Bluetit inter-process communication is completely based on D-Bus and implemented through the AirVPN–SUITE classes `DBusConnector`, `DBusResponse` and `DBusConnectorException`.

D-Bus configuration of Bluetit daemon is done during the installation process and the provided scripts will copy all the needed files in the proper directories. Bluetit *runs out-of-the-box* soon after the end of the installation process and no other configuration procedures are needed.

5.1 D-Bus Names

Bluetit conforms to the D-Bus specifications and, as such, defines its own D-Bus names in order to allow client to properly have an inter-process communication with the server. These names are required by D-Bus when the client is going to connect to Bluetit daemon and wants to call its public methods. Figure 5.1 shows all D-Bus names used by Bluetit and that must be referenced by the client.

5.2 D-Bus Configuration Files

The AirVPN–SUITE[1] installation package is distributed with both server and client D-Bus configuration unit files which are installed in the standard path, usually `/etc/systemd/system`.

Both configuration files define the access policy to Bluetit server and client, however the system administrator can modify them in order to suit system needs and policy. System administrators are strongly advised to carefully considering and pondering the rules defined in the D-Bus configuration files as they have a direct effect on system security. Not to mention, a bad or loose access policy may also result in a weak system protection therefore bringing the whole system to a critical condition. For this reason, the system administrator should carefully ponder how users or groups of users can have access to Bluetit daemon as this may grant unauthorized or "normal" users access to restricted data or parts of the system, including the risk of exploits, violation and breaching.

Bluetit has been designed in order to minimize exploit risks from within the daemon by limiting, as much as possible, the use of potentially weak and fragile constructs, including the call to external binaries, in particular the `shell`. Despite of this, and because of the indispensable Linux tools and architecture, some Bluetit internal functions and services require the calling of an external tool, in particular those

[1] AirVPN–SUITE package for Linux is available here: `https://airvpn.org/linux/suite`

D-Bus Interface Name: `org.airvpn.dbus`
Bluetit Server Bus Name: `org.airvpn.server`
Client Bus Name: `org.airvpn.client`
Bluetit Server Object Path Name: `/org/airvpn/server`
Client Object Path Name: `/org/airvpn/client`

Figure 5.1: D-Bus names used by the Bluetit daemon and client

related to the firewall services required for Bluetit's exclusive "Network Lock" feature, namely `iptables` and `nft`. All the external firewall tools used by Bluetit are called by using the standard C library function `execv()` and directly referring to their true path.

5.2.1 Server Configuration

Figure 5.2 shows the default D-Bus unit (configuration) file distributed with the AirVPN–SUITE package. The definition is rather simple and the only and sole owner of the Bluetit's server D-Bus name is the `root` user.

Moreover, it also defines a rule forbidding anyone, by default, the access to both the D-Bus interface name `"org.airvpn.dbus"` and Client D-Bus name `"org.airvpn.client"`. The policy for `"org.airvpn.client"` is properly defined in the client configuration file. By default, the server does not allow access to anyone but `root`.

5.2.2 Client Configuration

Figure 5.3 shows the default D-Bus unit (configuration) file and distributed with the AirVPN–SUITE package. As it can be seen, the default policy defines that a "client" is owned by root and all the users belonging to the `airvpn` user group. Moreover, they are both granted access to the server (that is, Bluetit daemon). All the other users in the system are explicitly denied accessing any of the Bluetit methods, interfaces or resources.

This access policy aims at providing a more reliable and solid security approach as access is granted only to users belonging to the `airvpn` user group. The system administrator should in fact be aware any user having access to the Bluetit D-Bus interface can actually start, stop and manage a VPN connection, that is can, as a matter of fact, alter the network layer, structure and security of the whole system.

5.3 Return Messages

Bluetit exposes to the allowed clients public methods in order to command specific operation or getting information about the connection and the AirVPN infrastructure. The whole communication between the Bluetit daemon and the client is done through the D-Bus service running in the system and, in particular, by making use of the `DBusConnector`, `DBusResponse` and `DBusConnectorException`.

All the public Bluetit D-Bus methods return a message and an associated data type describing the result of the requested operation. Each Bluetit public method returns a `DBusMessage` as provided by the D-Bus architecture and the caller needs to refer to this message in order to get and access the associated data.

The data associated to a `DBusMessage` can be retrieved by using the dedicated methods provided by the `DBusConnector` class, in particular `getInt()`, `getVector()`, `getArgs()` and `getResponse()`. For the specific message and value returned by each of the Bluetit methods, refer to the following sections.

```
<!DOCTYPE busconfig PUBLIC
 "-//freedesktop//DTD D-BUS Bus Configuration 1.0//EN"
 "http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd">
<busconfig>

  <policy user="root">
    <allow own="org.airvpn.server"/>
  </policy>

  <policy context="default">
    <deny send_interface="org.airvpn.dbus"/>
    <deny send_destination="org.airvpn.client"/>
  </policy>

</busconfig>
```

Figure 5.2: Default D-Bus configuration file for Bluetit daemon (server)

```
<!DOCTYPE busconfig PUBLIC
 "-//freedesktop//DTD D-BUS Bus Configuration 1.0//EN"
 "http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd">
<busconfig>

  <policy user="root">
    <allow own="org.airvpn.client"/>
    <allow send_interface="org.airvpn.dbus"/>
    <allow send_destination="org.airvpn.server"/>
  </policy>

  <policy group="airvpn">
    <allow own="org.airvpn.client"/>
    <allow send_interface="org.airvpn.dbus"/>
    <allow send_destination="org.airvpn.server"/>
  </policy>

  <policy context="default">
    <deny send_interface="org.airvpn.dbus"/>
    <deny send_destination="org.airvpn.server"/>
  </policy>

</busconfig>
```

Figure 5.3: Default D-Bus configuration file for Bluetit client

5.4 Public D-Bus Methods

Bluetit provides for a set of D-Bus method in order to allow the client to request information or run VPN related services, such as starting, pausing and stopping a VPN connection.

Each D-Bus method has its own interface – that is requires arguments and options, when needed, and always returns a data type – however their invocation and use is common to all of them. In general terms, the client calls a standard D-Bus method – either by using AirVPN–SUITE's `DBusConnector` class or D-Bus standard functions – then waits for a reply from D-Bus, if any, and finally processes the returned data.

In case a Bluetit method requires arguments, they must be provided always by means of a standard C++ vector object containing standard C++ string objects, that is a `std::vector<std::string>` object.

There are special cases in which a Bluetit D-Bus method – in particular D-Bus `"set_options"` method described in section 5.4.6 – may return a dataset containing a set of information relative to a specific group of homogeneous data. Datasets returned by Bluetit's D-Bus methods are always represented by a `DBusResponse` object as described in section 5.6 Response Dataset Identities.

5.4.1 version

Bluetit version

```
version
```

Returns full version information of the Bluetit daemon running in the system. It returns a `DBusResponse` object with no items and containing the version information in the response attribute.

Return:

DBusResponse.response: Bluetit version information: Bluetit name, version and release date.

Response
Bluetit name, Bluetit version, Bluetit release date

5.4.2 bluetit_status

Bluetit daemon status

```
bluetit_status
```

Returns the status code of Bluetit daemon,

Return:

DBusMessage.int: Bluetit integer status code

Bluetit Status Codes
1: Ready to accept commands and connections **2**: Connected to VPN **3**: VPN connection paused **4**: Bluetit dirty status. It means Bluetit has not successfully exited at the end of its last run or ended abnormally. **5**: Unreadable Bluetit resource directory or not found **6**: Initialization error. See Bluetit log for more information. **9**: Bluetit lock file not found. System has been probably tampered. **99**: Unknown or indeterminable status

5.4.3 openvpn_info

OpenVPN version

```
openvpn_info
```

Returns the version information of the OpenVPN infrastructure/library currently used by Bluetit

Return:

DBusResponse.response: OpenVPN version information

Response
OpenVPN version, platform and release information

5.4.4 openvpn_copyright

OpenVPN copyright notice

```
openvpn_copyright
```

Returns the copyright notice and information about the OpenVPN infrastructure/library currently used by Bluetit

Return:

DBusResponse.response: OpenVPN copyright information

Response
OpenVPN copyright information

5.4.5 reset_bluetit_options

Reset all Bluetit internal options

```
reset_bluetit_options
```

Resets all the Bluetit internal options to their default values

Return:

DBusResponse.response: String description of exit status

Response
Description of exit status

5.4.6 set_options

Sets Bluetit options

```
set_options
```

This method is used to send a set of options to the Bluetit daemon. It can be called for each option to be set or by sending the whole group and relative values in just one call. To see an example on how to call this method, refer to subsection 4.2.7 callMethodWithReply(). Specifically, this method can accept all the options currently available in `Goldcrest` client, both long and short versions. The options sent to Bluetit with this method must be specified without the dash or double dash. For example the `Goldcrest` long option `"-air-connect"` must be used in this method as `"air-connect"`, while the short option `"-C"` (cipher) must be used as `"C"`.

Arguments:

std::vector<std::string>: String vector of options and relative values to be set in Bluetit daemon. For a list of valid options and relative value, see Bluetit Users Manual distributed with AirVPN–SUITE (`README.md` file) or visit `https://airvpn.org/suite/readme`.

Return:

DBusResponse.response: String description of exit status

DBusResponse.items: Dataset associated to the option, when applicable. For more information about `DBusResponse` datasets, see 5.6 Response Dataset Identities

Response	Item 0	Item 1...*n*
Description of exit status	Dataset Identity	Dataset items

5.4.7 set_openvpn_profile

Sets the OpenVPN profile (configuration)

```
set_openvpn_profile
```

Sends and sets a generic OpenVPN profile (configuration file) to Bluetit to be used for a VPN connection. This method is used only to connect non-AirVPN servers and it is meant to allow the client to connect to any and generic OpenVPN server. It can also be used for connecting an AirVPN server in case the client wants to use its own configuration and does not want to use Bluetit built-in support to AirVPN infrastructure. To connect an AirVPN server and taking advantage of the full Bluetit integration and support to the AirVPN universe, the client is advised to use AirVPN's specific options as described in Bluetit Users Manual distributed with AirVPN–SUITE (`README.md` file) or visit `https://airvpn.org/suite/readme/`.

Arguments:

std::vector<std::string>: String vector containing the OpenVPN profiles. Each element of the vector contains just one OpenVPN profile and is represented in its usual profile (configuration) file format. The OpenVPN profile can be directly read from its file and directly assigned to the string. At the current version, the vector can contain just one OpenVPN profile and must be saved in the first element (0 index position).

Return:

DBusResponse.response: String description of exit status

Response
Description of exit status

5.4.8 start_connection

Starts an OpenVPN connection

```
start_connection
```

Starts an OpenVPN connection for the current OpenVPN profile set by the "set_openvpn_profile" method. It also enables the Network Lock, in case the client has used this option.

Return:

DBusResponse.response: String description of exit status

Response
Description of exit status

5.4.9 stop_connection

Stops a VPN connection

```
stop_connection
```

Stops the active VPN connection, either to AirVPN or generic OpenVPN server. It also disables the Network Lock, in case it has been enabled by the client. This method will however activate Bluetit persistent Network Lock in case the system administrator has set this option in Bluetit run control file.

Return:

DBusResponse.response: String description of exit status

Response
Description of exit status

5.4.10 pause_connection

Pauses a VPN connection

```
pause_connection
```

Pauses the active VPN connection, either to AirVPN or generic OpenVPN server. This method will however keep the Network Lock active in case it has been enabled by the client during connection. Moreover, the tunnel may be closed when the connection is paused according to the tunnel persist mode specified for the active connection or in Bluetit run control file.

Return:

DBusResponse.response: String description of exit status

Response
Description of exit status

5.4.11 resume_connection

Resumes a VPN connection

```
resume_connection
```

Resumes a previously paused VPN connection.

Return:

DBusResponse.response: String description of exit status

Response
Description of exit status

5.4.12 reconnect_connection

Reconnects a VPN connection

```
reconnect_connection
```

Reconnects (restart) the active VPN connection.

Return:

DBusResponse.response: String description of exit status

Response
Description of exit status

5.4.13 session_pause

Pauses the current VPN session

```
session_pause
```

Pauses the active and current VPN session, either to AirVPN or generic OpenVPN server. A session is a connection started and owned by another client or process, including Bluetit 's connect at boot feature. The use of this method depends on how Bluetit D-Bus interface and bus have been configured by the system administrator, that is whether concurrent access to the D-Bus is permitted or not. The default configuration provided with the AirVPN–SUITE allows just one client connection to Bluetit, that is grants "exclusive use and ownership" to the first client connecting to Bluetit. This means this method can be used, by default, only in case there is no other Bluetit client running. This method will however keep the Network Lock active in case it has been enabled by the client during connection. Moreover, the tunnel may be closed when the connection is paused according to the tunnel persist mode specified for the active connection or in Bluetit run control file.

Return:

DBusResponse.response: String description of exit status

Response
Description of exit status

5.4.14 session_resume

Resumes a paused VPN session

```
session_resume
```

Resumes a previously paused VPN session. A session is a connection started and owned by another client or process, including Bluetit's connect at boot feature. The use of this method depends on how Bluetit D-Bus interface and bus have been configured by the system administrator, that is whether concurrent access to the D-Bus is permitted or not. The default configuration provided with the AirVPN–SUITE allows just one client connection to Bluetit, that is grants "exclusive use and ownership" to the first client connecting to Bluetit. This means this method can be used, by default, only in case there is no other Bluetit client running. This method will however keep the Network Lock active in case it has been enabled by the client during connection. Moreover, the tunnel may be closed when the connection is paused according to the tunnel persist mode specified for the active connection or in Bluetit run control file.

Return:

DBusResponse.response: String description of exit status

Response
Description of exit status

5.4.15 session_reconnect

Reconnects the VPN session

```
session_reconnect
```

Reconnects (restart) the active VPN session. A session is a connection started and owned by another client or process, including Bluetit's connect at boot feature. The use of this method depends on how Bluetit D-Bus interface and bus have been configured by the system administrator, that is whether concurrent access to the D-Bus is permitted or not. The default configuration provided with the AirVPN–SUITE allows just one client connection to Bluetit, that is grants "exclusive use and ownership" to the first client connecting to Bluetit. This means this method can be used, by default, only in case there is no other Bluetit client running. This method will however keep the Network Lock active in case it has been enabled by the client during connection. Moreover, the tunnel may be closed when the connection is paused according to the tunnel persist mode specified for the active connection or in Bluetit run control file.

Return:

DBusResponse.response: String description of exit status

Response
Description of exit status

5.4.16 connection_stats

Gets VPN statistics

```
connection_stats
```

Gets the statistics information of the current and active VPN connection.

Return:

DBusResponse.response: String description of exit status
DBusResponse.item[0]: Connection statistics information

Response
`OK`: Data successfully retrieved
`ERROR`: Failed to retrieve data

Item 0
STATUS: Current Bluetit status
USER: User name associated to the OpenVPN connection
SERVER_HOST: Server host name
SERVER_PORT: Server port
SERVER_PROTO: Server protocol in use
SERVER_IP: Server IP address
VPN_IP4: VPN exit IPv4 address
VPN_IP6: VPN exit IPv6 address
GATEWAY_IPV4: IPv4 gateway address
GATEWAY_IPV6: IPv6 gateway address
CLIENT_IP: Client IP address
TUN_NAME: Tunnel device name
AIRVPN_SERVER_NAME: AirVPN server name
AIRVPN_SERVER_LOCATION: AirVPN server location
AIRVPN_SERVER_REGION: AirVPN server region
AIRVPN_SERVER_COUNTRY: AirVPN server country
AIRVPN_SERVER_COUNTRY_CODE: AirVPN server ISO country code
AIRVPN_SERVER_CONTINENT: AirVPN server continent
AIRVPN_SERVER_BANDWIDTH: AirVPN server effective current bandwidth in MBit/s
AIRVPN_SERVER_MAX_BANDWIDTH: AirVPN server maximum bandwidth in bytes
AIRVPN_SERVER_USERS: Number of users currently connected to the AirVPN server
AIRVPN_SERVER_LOAD: AirVPN server current load (percent)
AIRVPN_SERVER_WARNING_OPEN: Open warning message for AirVPN server
AIRVPN_SERVER_WARNING_CLOSED: Closed warning message for AirVPN server
AIRVPN_SERVER_TLS_CIPHERS: AirVPN server supported TLS ciphers
AIRVPN_SERVER_TLS_SUITE_CIPHERS: AirVPN server supported TLS suite ciphers
AIRVPN_SERVER_DATA_CIPHERS: AirVPN server supported data ciphers
AIRVPN_SERVER_SCORE: AirVPN server performance score (the higher, the better)
CONNECTION_TIME: Connection time in seconds
RATE_IN: Current input rate in bytes per second
BYTES_IN: Total input bytes count
RATE_OUT: Current output rate in bytes per second
BYTES_OUT: Total output bytes count
MAX_RATE_IN: Maximum recorded input rate in bytes per second
MAX_RATE_OUT: Maximum recorded output rate in bytes per second
TUN_BYTES_IN: Total tunnel input bytes count
TUN_BYTES_OUT: Total tunnel output bytes count
TUN_PACKETS_IN: Total tunnel input packets count
TUN_PACKETS_OUT: Total tunnel output packets count

Note: the AIRVPN_SERVER_* elements are valid only in case Bluetit is connected to an AirVPN server. In case Bluetit is connected to an AirVPN server, the element AIRVPN_SERVER_NAME always contains the server real name, whereas it is empty in case of connection to a generic OpenVPN file (that is, by using an OpenVPN profile or configuration file).

5.4.17 enable_network_lock

Enables the Network Lock

```
enable_network_lock
```

Enables the Network Lock by using the firewall infrastructure currently in use, either specified with the options or Bluetit run control file. In case Bluetit is configured to use a persistent Network Lock, this method has no effect.

Return:

DBusResponse.response: String description of exit status

Response
`OK`: Network Lock successfully enabled `ERROR`: Failed to enable Network Lock (including reason)

5.4.18 disable_network_lock

Disables the Network Lock

```
disable_network_lock
```

Disables the Network Lock by using the firewall infrastructure currently in use, either specified with the options or Bluetit run control file. In case Bluetit is configured to use a persistent Network Lock, this method has no effect.

Return:

DBusResponse.response: String description of exit status

Response
`OK`: Network Lock successfully disabled `ERROR`: Failed to disable Network Lock (including reason)

5.4.19 network_lock_status

Gets the Network Lock status

```
network_lock_status
```

Returns a description of the current Network Lock status

Return:

DBusResponse.response: String description of exit status

Response
Network Lock status description

5.4.20 recover_network

Performs a network recovery

```
recover_network
```

Tries to recover the system network and DNS configuration after an abnormal Bluetit termination

Return:

DBusResponse.response: String description of exit status

Response
Description of exit status

5.4.21 airvpn_set_key

Sets AirVPN key name

```
airvpn_set_key
```

Sets the AirVPN user key name to be used for an AirVPN connection.

Arguments:

std::vector<std::string>: String vector containing just one element (index 0) and representing the AirVPN user key name.

Return:

DBusResponse.response: String description of exit status

Response
`OK`: AirVPN user key name successfully set
`ERROR`: Invalid or empty user key name

5.4.22 airvpn_start_connection

Starts an AirVPN connection

```
airvpn_start_connection
```

Starts the connection to the AirVPN server according to the specific AirVPN options and modes set with `"set_options"` method.

Return:

DBusResponse.response: String description of exit status

Response
`OK`: AirVPN connection successfully started
`ERROR`: Failed to start AirVPN connection (see Bluetit log for more information)

5.4.23 event

Sends and event to the client

```
event
```

Sends an event to the client with the associated payload. This method is immediate and does not return any result. Refer to section 5.5 Bluetit Events for valid Bluetit event types and meaning.

Arguments:

std::vector<std::string>: String vector containing the event type and payload (see below)

Arguments
`0`: Event type
`1`: Event payload

5.4.24 log

Sends a log message to the client

```
log
```

Sends a log message to the client. This method is immediate and does not return any result.

Arguments:

std::vector<std::string>: String vector containing just one element (index 0) and representing the log message.

5.5 Bluetit Events

Bluetit can signal the client special conditions and statuses occurring while it is running by sending events. Bluetit events are not D-Bus events. For this specific purpose it is instead used the D-Bus `"event"` method which can be therefore conveniently received and processed in the client's D-Bus main loop or in a dedicated thread.

These events can optionally have an associated *payload* representing the event's own data, such a message or serialized data.

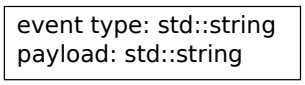

Figure 5.4: The structure of a Bluetit event

Figure 5.4 shows the structure of a Bluetit event. It is a simple `std::vector<std::string>` object made of two items, the first representing the event identification name and the second is its associated payload. The payload is optional and not all the events provide for a payload data. The two elements of a Bluetit event are both represented by a standard C++ `std::string`.

Bluetit events are to be considered immediate D-Bus methods and require no further action from the client, that is, no reply is needed to be sent to Bluetit. Event processing is required to be *"on-time"* as Bluetit events are not queued or buffered in any way. This means, in case the client does not get the event in the time it has been generated and sent, it is lost forever. This is also what happens for every D-Bus method or message and – for this reason – Bluetit events obey to D-Bus architecture and conventions.

Figure 5.5 shows an example on how to send an `"event_connected"` event to the client with a payload representing a descriptive reason of the event. As it can be seen, sending an event is equivalent to calling a D-Bus method without expecting a reply from the client.

Figure 5.6 shows an example on how to receive and process a Bluetit event as well as an example of a D-Bus loop using the `DBusConnector` class. As it can be seen, intercepting an event simply means to "pop" a D-Bus message from the queue and check whether it is "event" method. In that case the caller needs to read the arguments of the message (by using the `getArgs()` method) and then check what event type has been sent and possibly using the associated payload. The use of `getArgs()` method to retrieve data from the `DBusMessage` structure. As this is a native D-Bus data type, the associated data need to be retrieved by using the standard convention provided by the D-Bus specifications, therefore a reference to a `char *` and referenced by D-Bus data type `DBUS_TYPE_STRING`.

```
#include <dbusconnector.hpp>

DBusConnector *dbusConnector;
std::vector<std::string> dbusItems;

....

dbusItems.clear();

dbusItems.push_back("event_connected");
dbusItems.push_back("Successfully connected to the VPN server");

try
{
    dbusReply = dbusConnector->callMethod("org.airvpn.client",
    "/org/airvpn/client", "event", dbusItems);
}
catch(DBusConnectorException &e)
{
    // Error while sending the event
}
```

Figure 5.5: Example of sending a Bluetit event to the client

```
#include <iostream>
#include <dbusconnector.hpp>

DBusConnector *dbusConnector;
DBusMessage *dbusMessage;
char *event, *payload;
int done = false;

....

while(dbusConnector->readWriteDispatch() && done == false)
{
    while((dbusMessage = dbusConnector->popMessage()) != NULL && done == false)
    {
        if(dbusConnector->isMethod(dbusMessage, "event"))
        {
            if(dbusConnector->getArgs(dbusMessage, DBUS_TYPE_STRING, &event,
            DBUS_TYPE_STRING, &payload, DBUS_TYPE_INVALID))
            {
                if(strcmp(event, "event_connected") == 0)
                {
                    std::cout << "Connected: " << payload << std:endl;
                }
            }
        }

        dbusConnector->unreferenceMessage(dbusMessage);
    }
}
```

Figure 5.6: Example of receiving a Bluetit event from the daemon

To summarize it up, Bluetit event data are sent as standard C++ strings and are received by the client as D-Bus data types.

5.5.1 event_end_of_session

Terminates the client session

```
event_end_of_session
```

Terminates the current client session, logs the AirVPN out and resets all internal settings to their default values, therefore preparing Bluetit to accept and start a new client session

Arguments:

payload: Empty

5.5.2 event_connected

Connection established with server

```
event_connected
```

Connection established with the requested OpenVPN or AirVPN server

Arguments:

payload: Empty

5.5.3 event_disconnected

Connection with server terminated

```
event_disconnected
```

Connection with the requested OpenVPN or AirVPN server is terminated and closed

Arguments:

payload: Empty

5.5.4 event_pause

Connection with server paused

```
event_disconnected
```

Connection with the requested OpenVPN or AirVPN server is paused

Arguments:

payload: Empty

5.5.5 event_resume

Connection with server resumed

```
event_resume
```

Connection with OpenVPN or AirVPN server has been successfully resumed after pause

Arguments:

payload: Empty

5.5.6 event_error

Error condition

```
event_error
```

An error occurred within Bluetit activity, including connection status

Arguments:

payload: Error description

5.6 Response Dataset Identities

Bluetit is mainly driven by sending valid options to it and for which it responds with specific data or starts a required task. There are special cases in which a `DBusMessage` returns a `DBusResponse` object representing a dataset about a particular set of data and information.

Response datasets are always triggered by a specific Bluetit option and they are qualified with their identifier contained in the `DBusRespose`'s response attribute. The "versatility" of `DBusResponse`'s response allows a double use, that is to pass to the caller the result of a D-Bus method's task and to identify a dataset.

Figure 5.7 shows the structure of a possible Bluetit dataset. The dataset identifier, that is the tag qualifying the whole dataset, is represented by the `response` attribute of `DBusResponse` object, whereas the associated data are represented by the object's items. This means a dataset always has a response identifier and at least one item representing the associated data.

In order to receive a Bluetit dataset, the client needs to send the relative options to the daemon and by using the D-Bus method `set_options`, described in section 5.4.6.

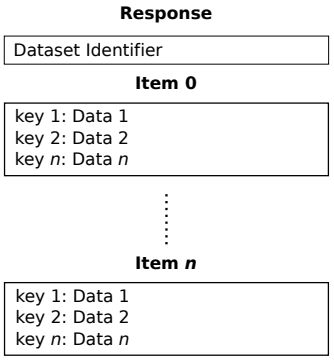

Figure 5.7: The structure of a Bluetit dataset

Figure 5.8 shows how to properly request a Bluetit dataset, in this case about the AirVPN server Orion. As it can be seen, the client simply needs to append the proper options to the `set_options` method's argument and then to call it. The Bluetit daemon will then reply with the corresponding dataset about the requested data.

The client is therefore requested to properly get the relative D-Bus reply and to process the dataset according to its needs. This can be conveniently made in the D-Bus main loop of the client or in dedicated thread or function. Figure 5.9 shows how to receive and evaluate a Bluetit dataset in response to the example shown in figure 5.8.

5.6.1 airvpn_server_info

Information about an AirVPN server

```
airvpn_server_info
```

```
#include <iostream>
#include <vector>
#include <dbusconnector.hpp>

DBusConnector *dbusConnector = nullptr;
DBusMessage *dbusReply = nullptr;
std::vector<std::string> dbusItems;

....

dbusItems.clear();

dbusItems.push_back("air-info");
dbusItems.push_back("air-server");
dbusItems.push_back("orion");

dbusReply = dbusConnector->callMethodWithReply("org.airvpn.server",
"/org/airvpn/server", "set_options", dbusItems);

if(dbusReply == nullptr)
{
    // Error
}
```

Figure 5.8: Example of requesting a Bluetit dataset about a specific AirVPN server

Complete information about an AirVPN server

Return:

DBusResponse.response: `"airvpn_server_info"` in case of success; Error description otherwise

DBusResponse.item[0]: AirVPN server information

Required options and sequence
0: air-info
1: air-server
2: <server full name>

Returned dataset:

Response
airvpn_server_info

```
#include <iostream>
#include <dbusconnector.hpp>

DBusConnector *dbusConnector;
DBusMessage *dbusReply;
DBusResponse *dbusResponse;
DBusResponse::Item item;
std::string response;

....

dbusResponse = dbusConnector->getResponse(dbusReply);

response = dbusResponse->getResponse();

if(response == "airvpn_server_info" && dbusResponse->rows() > 0)
{
    // Process server information

    item = dbusResponse->getItem(0);

    std::cout  << "Server name: " << dbusResponse->getItemValue(item, "name")
    << std::endl;
}
```

Figure 5.9: Example of processing a Bluetit dataset about a specific AirVPN server

Item 0
`name`: AirVPN server name
`country_code`: ISO code of AirVPN server country
`country`: AirVPN server country name
`location`: AirVPN server geographical location
`bandwidth`: AirVPN server current bandwidth in MBit/s
`effective_bandwidth`: AirVPN server effective bandwidth in MBit/s
`max_bandwidth`: AirVPN server maximum bandwidth in bytes
`users`: Number of users currently connected to the AirVPN server
`supports_ipv4`: AirVPN server support for IPv4 (yes, no)
`supports_ipv6`: AirVPN server support for IPv6 (yes, no)
`open_status`: Open warning message for AirVPN server
`close_status`: Close warning message for AirVPN server
`load`: AirVPN server current load (percent)
`tls_ciphers`: AirVPN server supported TLS ciphers
`tls_suite_ciphers`: AirVPN server supported TLS suite ciphers
`data_ciphers`: AirVPN server supported data ciphers
`score`: AirVPN server performance score (the higher, the better)
`available`: AirVPN server availability (yes, no)

5.6.2 airvpn_server_list

List of AirVPN servers

```
airvpn_server_list
```

Represents a list of AirVPN servers satisfying the specified pattern. A server is included in the list in case the pattern is contained in the server name, country ISO code or country description. In case the pattern is equal to `"ALL"`, a list of all available AirVPN servers is returned.

Return:

DBusResponse.response: `"airvpn_server_list"` in case of success; Error description otherwise

DBusResponse.item[0...*n*]: AirVPN server information. Each entry (server) is defined in its specific `DBusResponse` item.

Required options and sequence
`0: air-list`
`1: air-server`
`2: <pattern|ALL>`

Returned dataset:

Response
`airvpn_server_info`

Item 0...*n*
`name`: AirVPN server name
`country_code`: ISO code of AirVPN server country
`country`: AirVPN server country name
`location`: AirVPN server geographical location
`bandwidth`: AirVPN server current bandwidth in MBit/s
`effective_bandwidth`: AirVPN server effective bandwidth in MBit/s
`max_bandwidth`: AirVPN server maximum bandwidth in bytes
`users`: Number of users currently connected to the AirVPN server
`supports_ipv4`: AirVPN server support for IPv4 (yes, no)
`supports_ipv6`: AirVPN server support for IPv6 (yes, no)
`open_status`: Open warning message for AirVPN server
`close_status`: Close warning message for AirVPN server
`load`: AirVPN server current load (percent)
`tls_ciphers`: AirVPN server supported TLS ciphers
`tls_suite_ciphers`: AirVPN server supported TLS suite ciphers
`data_ciphers`: AirVPN server supported data ciphers
`score`: AirVPN server performance score (the higher, the better)
`available`: AirVPN server availability (yes, no)

5.6.3 airvpn_country_info

Information about an AirVPN country

```
airvpn_country_info
```

Complete information about an AirVPN country in which servers are available

Return:

DBusResponse.response: `"airvpn_country_info"` in case of success; Error description otherwise

DBusResponse.item[0]: Country information

Required options and sequence
`0: air-info`
`1: air-country`
`2: <country full name|country ISO code>`

Returned dataset:

Response
`airvpn_country_info`

Item 0
`country_iso_code`: Country ISO code
`country_name`: Country full name
`servers`: Number of AirVPN servers available in the country
`users`: Number of users currently connected to the country
`bandwidth`: Country current bandwidth in MBit/s
`max_bandwidth`: Country maximum bandwidth in bytes

5.6.4 airvpn_country_list

List of AirVPN countries

```
airvpn_country_list
```

Represents a list of countries where AirVPN servers are available and satisfying the specified pattern.

A country is included in the list in case the pattern is contained in its name or ISO code. In case the pattern is equal to `"ALL"`, a list of all available countries in the AirVPN's universe is returned.

Return:

DBusResponse.response: `"airvpn_country_list"` in case of success; Error description otherwise

DBusResponse.item[0...*n*]: Country information. Each entry (country) is defined in its specific `DBusResponse` item.

Required options and sequence
0: air-list
1: air-country
2: <country full name|country ISO code|ALL>

Returned dataset:

Response
airvpn_country_list

Item 0...*n*
`country_iso_code`: Country ISO code
`country_name`: Country full name
`servers`: Number of AirVPN servers available in the country
`users`: Number of users currently connected to the country
`bandwidth`: Country current bandwidth in MBit/s
`max_bandwidth`: Country maximum bandwidth in bytes

5.6.5 airvpn_key_list

List of keys for AirVPN user

```
airvpn_key_list
```

Represents a list of user keys (device profiles) associated to the specified AirVPN user name and password.

Return:

DBusResponse.response: `"airvpn_key_list"` in case of success; Error description otherwise

DBusResponse.item[0...*n*]: User key information. Each entry (key) is defined in its specific `DBusResponse` item.

Required options and sequence
0: air-user
1: <AirVPN user name>
2: air-password
3: <AirVPN user password>
4: air-key-list

Returned dataset:

Response
airvpn_key_list

Item 0...*n*
`key`: Key name

5.6.6 airvpn_save

Configuration data for saving

airvpn_save

This dataset represents a document or configuration to be saved by the client in a local file or used otherwise. It basically represents two types of documents: an OpenVPN configuration (profile) about an AirVPN server or country as well as the OpenVPN certificates and keys for a specific user. The document type is specified in the `type` element of the returned item.

Return:

DBusResponse.response: `"airvpn_save"` in case of success; Error description otherwise
DBusResponse.item[0]: Document information

Requesting the OpenVPN configuration for an AirVPN server

Required options and sequence
0: air-user
1: <AirVPN user name>
2: air-password
3: <AirVPN user password>
4: air-save
5: <file name>
6: air-server
7: <AirVPN server name>

Returned dataset:

Response
airvpn_save

Item 0
type: "profile for server"
user: AirVPN user name
name: AirVPN server name
file_name: <provided file name>
content: <OpenVPN configuration>

..

Requesting the OpenVPN configuration for an AirVPN server and specific AirVPN user key

Required options and sequence
0: air-user
1: <AirVPN user name>
2: air-password
3: <AirVPN user password>
4: air-save
5: <file name>
6: air-server
7: <AirVPN server name>
8: air-key
9: <user key name>

Returned dataset:

Response
airvpn_save

Item 0
type: "profile for server"
user: AirVPN user name
name: AirVPN server name
file_name: <provided file name>
content: <OpenVPN configuration>

Requesting the OpenVPN configuration for an AirVPN country

Required options and sequence
0: air-user
1: <AirVPN user name>
2: air-password
3: <AirVPN user password>
4: air-save
5: <file name>
6: air-country
7: <AirVPN country name|AirVPN country ISO code>

Returned dataset:

Response
airvpn_save

Item 0
type: "profile for country"
user: AirVPN user name
name: AirVPN country name
file_name: <provided file name>
content: <OpenVPN configuration>

Requesting the OpenVPN configuration for an AirVPN country and specific AirVPN user key

Required options and sequence
0: air-user
1: <AirVPN user name>
2: air-password
3: <AirVPN user password>
4: air-save
5: <file name>
6: air-country
7: <AirVPN country name|AirVPN country ISO code>
8: air-key
9: <user key name>

Returned dataset:

Response
airvpn_save

Item 0
type: "profile for country"
user: AirVPN user name
name: AirVPN country name
file_name: <provided file name>
content: <OpenVPN configuration>

Requesting the OpenVPN certificates and keys for an AirVPN user

Required options and sequence
0: air-user
1: <AirVPN user name>
2: air-password
3: <AirVPN user password>
4: air-save
5: <file name>
6: air-key
7: <user key name>

Returned dataset:

Response
airvpn_save

Item 0
type: "key"
user: AirVPN user name
name: Key name
file_name: <provided file name>
content: <OpenVPN certificates/keys>