WIKIPEDIA

D-Bus

In computing, **D-Bus** (short for "**Desktop Bus**"[4]) is a message-oriented middleware mechanism that allows communication between multiple processes running concurrently on the same machine.[5][6] D-Bus was developed as part of the freedesktop.org project, initiated by Havoc Pennington from Red Hat to standardize services provided by Linux desktop environments such as GNOME and KDE.[7][8]

The freedesktop.org project also developed a free and open-source software library called libdbus, as a reference implementation of the specification. This library should not be confused with D-Bus itself, as other implementations of the D-Bus specification also exist, such as GDBus (GNOME),[9] QtDBus (Qt/KDE),[10] dbus-java[11] and sd-bus (part of systemd).[12]

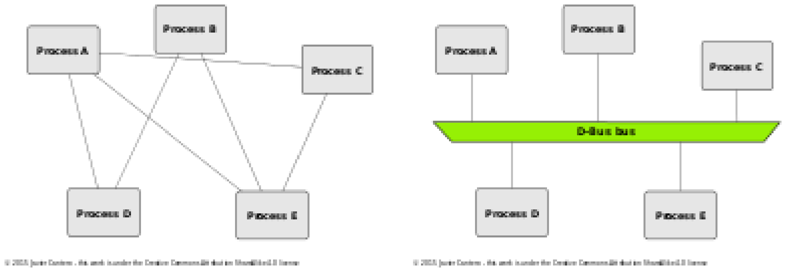
D-Bus

Developer(s)	Red Hat
Initial release	November 2006
Stable release	1.14.0 / February 28, 2022[1]
Preview release	1.15.0 / February 28, 2022[2]
Repository	cgit.freedesktop.org/dbus/dbus/ (https://cgit.freedesktop.org/dbus/dbus/)
Written in	C
Operating system	Cross-platform
Type	IPC daemon Linux on the desktop
License	GPLv2+ or AFL 2.1[3]
Website	www.freedesktop.org/wiki/Software/dbus (http://www.freedesktop.org/wiki/Software/dbus)

Contents

Overview

D-Bus is an inter-process communication (IPC) mechanism initially designed to replace the software component communications systems used by the GNOME and KDE Linux desktop environments (CORBA and DCOP respectively).[13][14] The components of these desktop environments are normally distributed in many processes, each one providing only a few —usually one— *services*. These services may be used by regular client applications or by other components of the desktop environment to perform their tasks.



Processes without D-Bus

The same processes with D-Bus

Large groups of cooperating processes demand a dense mesh of individual communication channels (using one-to-one IPC methods) between them. D-Bus simplifies the IPC requirements with one single shared channel.

Due to the large number of processes involved —adding up processes providing the services and clients accessing them— establishing one-to-one IPC communications between all of them becomes an inefficient and quite unreliable approach. Instead, D-Bus provides a software-bus abstraction that gathers all the communications between a group of processes over a single shared virtual channel.[6] Processes connected to a bus do not know how it is internally implemented, but D-Bus specification guarantees that all processes connected to the bus can communicate with each other through it.

Linux desktop environments take advantage of the D-Bus facilities by instantiating multiple buses, notably:[15][6][16]

- a single **system bus**, available to all users and processes of the system, that provides access to system services (i.e. services provided by the operating system and also by any system daemons)
- a **session bus** for each user login session, that provides desktop services to user applications in the same desktop session, and allows the integration of the desktop session as a whole

A process can connect to any number of buses, provided that it has been granted access to them. In practice, this means that any user process can connect to the system bus and to its current session bus, but not to another user's session buses, or even to a different session bus owned by the same user. The latter restriction may change in the future if all user sessions are combined into a single user bus.[17]

D-Bus provides additional or simplifies existing functionality to the applications, including information-sharing, modularity and privilege separation. For example, information on an incoming voice-call received through Bluetooth or Skype can be propagated and interpreted by any currently-running music player, which can react by muting the volume or by pausing playback until the call is finished.[18]

D-Bus can also be used as a framework to integrate different components of a user application. For instance, an office suite can communicate through the session bus to share data between a word processor and a spreadsheet.

D-Bus specification

Bus model

Every connection to a bus is identified in the context of D-Bus by what is called a *bus name*.[5] A bus name consists of two or more dot-separated strings of letters, digits, dashes, and underscores. An example of a valid bus name is `org.freedesktop.NetworkManager`.[6]

When a process sets up a connection to a bus, the bus assigns to the connection a special bus name called *unique connection name*.[16][6] Bus names of this type are immutable—it's guaranteed they won't change as long as the connection exists—and, more importantly, they can't be reused during the bus lifetime.[5][16][6] This means that no other connection to that bus will ever have assigned such unique connection name, even if the same process closes down the connection to the bus and creates a new one. Unique connection names are easily recognizable because they start with the—otherwise forbidden—colon character.[16][6] An example of a unique connection name is `:1.1553` (the characters after the colon have no particular meaning[16]).

A process can ask for additional bus names for its connection,[16] provided that any requested name is not already being used by another connection to the bus. In D-Bus parlance, when a bus name is assigned to a connection, it is said the connection *owns* the bus name.[5][16] In that sense, a bus name can't be owned by two connections at the same time, but, unlike unique connection names, these names can be reused if they are available: a process may reclaim a bus name released —purposely or not— by another process.[5][6]

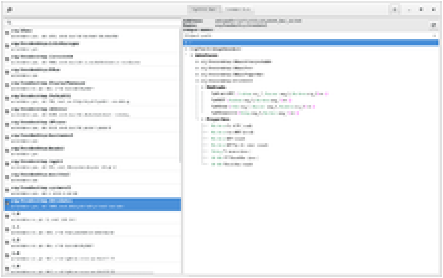
The idea behind these additional bus names, commonly called *well-known names*, is to provide a way to refer to a service using a prearranged bus name.[16][6] For instance, the service that reports the current time and date in the system bus lies in the process whose connection owns the `org.freedesktop.timedate1` bus name, regardless of which process it is.

Bus names can be used as a simple way to implement single-instance applications (second instances detect that the bus name is already taken).[16] It can also be used to track a service process lifecycle, since the bus sends a notification when a bus name is released due to a process termination.[16]

Object model

Because of its original conception as a replacement for several component oriented communications systems, D-Bus shares with its predecessors an object model in which to express the semantics of the communications between clients and services. The terms used in the D-Bus object model mimic those used by some object oriented programming languages. That doesn't mean that D-Bus is somehow limited to OOP languages —in fact, the most used implementation (`libdbus`) is written in C, a procedural programming language.

In D-Bus, a process offers its services by exposing *objects*. These objects have *methods* that can be invoked, and *signals* that the object can emit.[16] Methods and signals are collectively referred to as the *members* of the object.[5] Any client connected to the bus can interact with an object by using its methods, making requests or commanding the object to perform actions.[16] For instance, an object representing a time service can be queried by a client using a method that returns the current date and time. A client can also listen to signals that an object emits when its state changes due to certain events, usually related to the underlying service. An example would be when a service that manages hardware devices —such as USB or network drivers— signals a "new hardware device added" event. Clients should instruct the bus that they are interested in receiving certain signals from a particular object, since a D-Bus bus only passes signals to those processes with a registered interest in them.[6]


Browsing the existing bus names, objects, interfaces, methods and signals in a D-Bus bus using D-Feet

A process connected to a D-Bus bus can request it to *export* as many D-Bus objects as it wants. Each object is identified by an *object path*, a string of numbers, letters and underscores separated and prefixed by the slash character, called that because of their resemblance to Unix filesystem paths.[5][16] The object path is selected by the requesting process, and must be unique in the context of that bus connection. An example of a valid object path is `/org/kde/kspread/sheets/3/cells/4/5`.[16] However, it's not enforced —but also not discouraged— to form hierarchies within object paths.[6] The particular naming convention for the objects of a service is entirely up to the developers of such service, but many developers choose to namespace them using the reserved domain name of the project as a prefix (e.g. `/org/kde`).[16]

Every object is inextricably associated to the particular bus connection where it was exported, and, from the D-Bus point of view, only lives in the context of such connection. Therefore, in order to be able to use a certain service, a client must indicate not only the object path providing the desired service, but also the bus name under which the service process is connected to the bus.[5] This in turn allows that several processes connected to the bus can export different objects with identical object paths unambiguously.

An *interface* specifies members —methods and signals— that can be used with an object.[16] It is a set of declarations of methods (including its passing and returning parameters) and signals (including its parameters) identified by a dot-separated name resembling the Java language interfaces notation.[16][6] An example of a valid interface name is `org.freedesktop.Introspectable`.[6] Despite their similarity, interface names and bus names should not be mistaken. A D-Bus object can *implement* several interfaces, but at least must implement one, providing support for every method and signal defined by it. The combination of all interfaces implemented by an object is called the object *type*.[5][16]

When using an object, it's a good practice for the client process to provide the member's interface name besides the member's name, but is only mandatory when there is an ambiguity caused by duplicated member names available from different interfaces implemented by the object[5][16] —otherwise, the selected member is undefined or erroneous. An emitted signal, on the other hand, must always indicate to which interface it belongs.

The D-Bus specification also defines several standard interfaces that objects may want to implement in addition to its own interfaces.[15] Although technically optional, most D-Bus service developers choose to support them in their exported objects since they offer important additional features to D-Bus clients, such as introspection.[6] These standard interfaces are:[15][6]

- `org.freedesktop.DBus.Peer`: provides a way to test if a D-Bus connection is alive.[6]
- `org.freedesktop.DBus.Introspectable`: provides an introspection mechanism by which a client process can, at run-time, get a description (in XML format) of the interfaces, methods and signals that the object implements.[16][15]

- `org.freedesktop.DBus.Properties`: allows a D-Bus object to expose the underlying native object properties or attributes, or simulate them if it doesn't exist.[15]
- `org.freedesktop.DBus.ObjectManager`: when a D-Bus service arranges its objects hierarchically, this interface provides a way to query an object about all sub-objects under its path, as well as their interfaces and properties, using a single method call.[15]

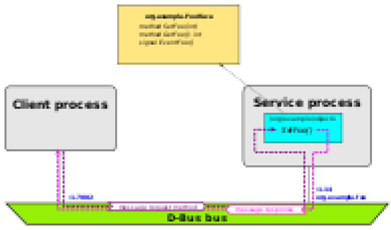
The D-Bus specification defines a number of administrative bus operations (called "bus services") to be performed using the `/org/freedesktop/DBus` object that resides in the `org.freedesktop.DBus` bus name.[15] Each bus reserves this special bus name for itself, and manages any requests made specifically to this combination of bus name and object path. The administrative operations provided by the bus are those defined by the object's interface `org.freedesktop.DBus`. These operations are used for example to provide information about the status of the bus,[5] or to manage the request and release of additional *well-known* bus names.[15][6]

Communications model

D-Bus was conceived as a generic, high-level inter-process communication system. To accomplish such goals, D-Bus communications are based on the exchange of *messages* between processes instead of "raw bytes".[5][16] D-Bus messages are high-level discrete items that a process can send through the bus to another connected process. Messages have a well-defined structure (even the types of the data carried in their payload are defined), allowing the bus to validate them and to reject any ill-formed message. In this regard, D-Bus is closer to an RPC mechanism than to a classic IPC mechanism, with its own type definition system and its own marshaling.[5]

The bus supports two modes of interchanging messages between a client and a service process[5]:

- One-to-one request-response: This is the way for a client to invoke an object's method. The client sends a message to the service process exporting the object, and the service in turn replies with a message back to the client process.[16] The message sent by the client must contain the object path, the name of the invoked method (and optionally the name of its interface), and the values of the input parameters (if any) as defined by the object's selected interface. The reply message carries the result of the request, including the values of the output parameters returned by the object's method invocation, or *exception* information if there was an error.[5][16]
- Publish/subscribe: This is the way for an object to announce the occurrence of a signal to the interested parties. The object's service process broadcasts a message that the bus passes only to the connected clients subscribed to the object's signal.[16] The message carries the object path, the name of the signal, the interface to which the signal belongs, and also the values of the signal's parameters (if any). The communication is one-way: there are no response messages to the original message from any client process, since the sender knows neither the identities nor the number of the recipients.[5][16]



Example of one-to-one request-response message exchange to invoke a method over D-Bus. Here the client process invokes the `SetFoo()` method of the `/org/example/object1` object from the service process named `org.example.foo` (or `:1.14`) in the bus.

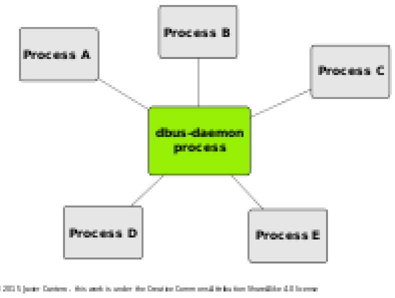
Every D-Bus message consists of a header and a body.[16] The header is formed by several fields that identify the type of message, the sender, as well as information required to deliver the message to its recipient (destination bus name, object path, method or signal name, interface name, etc.).[16][15] The body contains the data payload that the receiver process interprets —for instance the input or output arguments. All the data is encoded in a well known binary format called the *wire format* which supports the serialization of various types, such as integers and floating-point numbers, strings, compound types, and so on,[15] also referred to as marshaling.

The D-Bus specification defines the wire protocol: how to build the D-Bus messages to be exchanged between processes within a D-Bus connection. However, it does not define the underlying transport method for delivering these messages.

Internals

Most existing D-Bus implementations follow the architecture of the reference implementation. This architecture consists of two main components:[5]

- a point-to-point communications library that implements the D-Bus wire protocol in order to exchange messages between two processes. In the reference implementation this library is **`libdbus`**. In other implementations `libdbus` may be wrapped by another higher-level library, language binding, or entirely replaced by a different standalone implementation that serves the same purpose.[19] This library only supports one-to-one communications between two processes.[16]
- a special daemon process that plays the bus role and to which the rest of the processes connect using any D-Bus point-to-point communications library. This process is also known as the *message bus daemon*,[18] since it is responsible for routing messages from any process connected to the bus to another. In the reference implementation this role is performed by **`dbus-daemon`**, which itself is built on top of `libdbus`. Another implementation of the message bus daemon is **`dbus-broker`**, which is built on top of **`sd-bus`**.

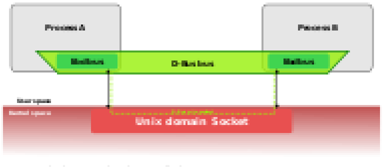


A `dbus-daemon` process acting as a D-Bus message bus daemon. Every process connected to the bus keeps one D-Bus connection with it.

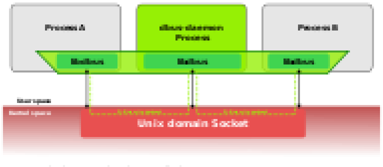
The `libdbus` library (or its equivalent) internally uses a native lower-level IPC mechanism to transport the required D-Bus messages between the two processes in both ends of the D-Bus connection. D-Bus specification doesn't mandate which particular IPC transport mechanisms should be available to use, as it's the communications library that decides what transport methods it supports. For instance, in Unix-like operating systems such as Linux `libdbus` typically uses Unix domain sockets as the underlying transport method, but it also supports TCP sockets.[5][16]

The communications libraries of both processes must agree on the selected transport method and also on the particular channel used for their communication. This information is defined by what D-Bus calls an *address*.[6][16] Unix-domain sockets are filesystem objects, and therefore they can be identified by a filename, so a valid address would be `unix:path=/tmp/.hiddensocket`.[5][15] Both processes must pass the same address to their respective communications libraries to establish the D-Bus connection between them. An address can also provide additional data to the communications library in the form of comma-separated `key=value` pairs.[6][15] This way, for example, it can provide authentication information to a specific type of connection that supports it.

When a message bus daemon like `dbus-daemon` is used to implement a D-Bus bus, all processes that want to connect to the bus must know the *bus address*, the address by which a process can establish a D-Bus connection to the central message bus process.[5][16] In this scenario, the message bus daemon selects the bus address and the remainder processes must pass that value to their corresponding `libdbus` or equivalent libraries. `dbus-daemon` defines a different bus address for every bus instance it provides. These addresses are defined in the daemon's configuration files.



Process A and B have a one-to-one D-Bus connection using `libdbus` over a Unix domain socket. They can use it to exchange messages directly.[20] In this scenario bus names are not required.[16]



Process A and B both connected to a `dbus-daemon` using `libdbus` over a Unix domain socket. They can exchange messages sending them to the message bus process, which in turn will deliver the messages to the appropriate process. In this scenario bus names are mandatory to identify the destination process.

Two processes can use a D-Bus connection to exchange messages directly between them,[20] but this is not the way in which D-Bus is normally intended to be used. The usual way is to always use a message bus daemon (i.e. `dbus-daemon`) as a communications central point to which each process should establish its point-to-point D-Bus connection. When a process —client or service— sends a D-Bus message, the message bus process receives it in the first instance and delivers it to the appropriate recipient. The message bus daemon may be seen as a hub or router in charge of getting each message to its destination by repeating it through the D-Bus connection to the recipient process.[16] The recipient process is determined by the destination bus name in the message's header field,[15] or by the subscription information to signals maintained by the message bus daemon in the case of signal propagation messages.[6] The message bus daemon can also produce its own messages as a response to certain conditions, such as an error message to a process that sent a message to a nonexistent bus name.[16]

`dbus-daemon` improves the feature set already provided by D-Bus itself with additional functionality. For example, *service activation* allows automatic starting of services when needed —when the first request to any bus name of such service arrives at the message bus daemon.[5] This way, service processes neither need to be launched during the system initialization or user initialization stage nor need they consume memory or other resources when not being used. This feature was originally implemented using setuid helpers,[21] but nowadays it can also be provided by systemd's service activation framework. Service activation is an important feature that facilitates the management of the process lifecycle of services (for example when a desktop component should start or stop).[16]

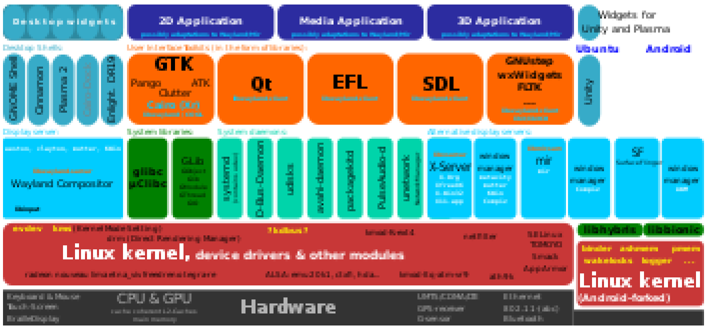
History and adoption

D-Bus was started in 2002 by Havoc Pennington, Alex Larsson (Red Hat) and Anders Carlsson.[8] The version 1.0 —considered API stable— was released in November 2006.[22][23]

Heavily influenced by the DCOP system used by versions 2 and 3 of KDE, D-Bus has replaced DCOP in the KDE 4 release.[23][24] An implementation of D-Bus supports most POSIX operating systems, and a port for Windows exists. It is used by Qt 4 and later by GNOME. In GNOME it has gradually replaced most parts of the earlier Bonobo mechanism. It is also used by Xfce.

One of the earlier adopters was the (nowadays deprecated) Hardware Abstraction Layer. HAL used D-Bus to export information about hardware that has been added to or removed from the computer.[8]

The usage of D-Bus is steadily expanding beyond the initial scope of desktop environments to cover an increasing amount of system services. For instance, the NetworkManager network daemon, BlueZ bluetooth stack and PulseAudio sound server use D-Bus to provide part or all of their services. systemd uses the D-Bus wire protocol for communication between `systemctl` and systemd, and is also promoting traditional system daemons to D-Bus services, such as logind.[25] Another heavy user of D-Bus is Polkit, whose policy authority daemon is implemented as a service connected to the system bus.[26]



The `dbus-daemon` plays a significant role in modern Linux graphical desktop environments.

Implementations

libdbus

Although there are several implementations of D-Bus, the most widely used is the reference implementation *libdbus*, developed by the same freedesktop.org project that designed the specification. However, libdbus is a low-level implementation that was never meant to be used directly by application developers, but as a reference guide for other reimplementations of D-Bus (such as those included in standard libraries of desktop environments, or in programming language bindings). The freedesktop.org project itself recommends applications authors to "use one of the higher level bindings or implementations" instead.[27] The predominance of libdbus as the most used D-Bus implementation caused the terms "D-Bus" and "libdbus" to be often used interchangeably, leading to confusion.

GDBus

GDBus[9] is an implementation of D-Bus based on GIO streams included in GLib, aiming to be used by GTK+ and GNOME. GDBus is not a wrapper of libdbus, but a complete and independent reimplementation of the D-Bus specification and protocol.[28] MATE Desktop[29] and Xfce (version 4.14), which are also based on GTK+ 3, also use GDBus.

QtDBus

QtDBus[10] is an implementation of D-Bus included in the Qt library since its version 4.2. This component is used by KDE applications, libraries and components to access the D-Bus services available in a system.

sd-bus

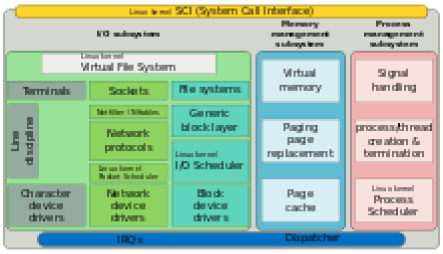
In 2013, the systemd project rewrote libdbus in an effort to simplify the code,[30] but it also resulted in a significant increase of the overall D-Bus performance. In preliminary benchmarks, BMW found that the systemd's D-Bus library increased performance by 360%.[31] By version 221 of systemd, the sd-bus API was declared stable.[32]

libnih-dbus

The libnih project provides a light-weight "standard library" of C support for D-Bus. Additionally, it has good support for cross compiling.

kdbus

kdbus was a project that aimed to reimplement D-Bus as a kernel-mediated peer-to-peer inter-process communication mechanism. Beside performance improvements, kdbus would have advantages arising from other Linux kernel features such as namespaces and auditing,[31][35] security from the kernel mediating, closing race conditions, and allowing D-Bus to be used during boot and shutdown (as needed by systemd).[36] kdbus inclusion in the Linux kernel proved controversial,[37] and was dropped in favor of BUS1 (https://bus1.org/), as a more generic inter-process communication.[38]



kdbus is implemented as a character device driver.[33][34] All communication between processes take place over special character device nodes in `/dev/kdbus` (cf. devfs).

zbus

zbus (https://gitlab.freedesktop.org/dbus/zbus) is a native Rust library for D-Bus. Its main strengths are its macros (https://docs.rs/zbus_macros/latest/zbus_macros/) that make communication with services and implementation of services, extremely easy and simple.

Protocol::DBus

Protocol::DBus (https://metacpan.org/pod/Protocol::DBus) is a native Perl D-Bus client.

Language bindings

Several programming language bindings for D-Bus have been developed,[39] such as those for Java, C#, Ruby, and Perl.

See also

- Linux on the desktop
- Common Language Infrastructure
- Common Object Request Broker Architecture
- Component Object Model
- Distributed Component Object Model
- Foreign function interface
- Java remote method invocation
- Remote procedure call
- XPCOM

References

1. "D-Bus 1.14.x changelog" (https://cgit.freedesktop.org/dbus/dbus/tree/NEWS?h=dbus-1.14). Retrieved 30 April 2022.
2. "NEWS file for current branch" (https://cgit.freedesktop.org/dbus/dbus/tree/NEWS?h=master). Retrieved 30 April 2022.
3. Havoc's Blog July, 2007 (http://blog.ometer.com/2007/07/17/gpl--afl/)
4. Ward, Brian (2004). "14: A brief survey of the Linux desktop". *How Linux Works: What Every Superuser Should Know* (https://books.google.com/books?id=fP5WBQAAQBAJ) (2 ed.). San Francisco: No Starch Press (published 2014). p. 305. ISBN 9781593275679. Retrieved 2016-11-07. "One of the most important developments to come out of the Linux desktop is the Desktop Bus (D-Bus), a message-passing system. D-Bus is important because it serves as an interprocess communication mechanism that allows desktop applications to talk to each other [...]."
5. Vermeulen, Jeroen (14 Jul 2013). "Introduction to D-Bus" (http://www.freedesktop.org/wiki/IntroductionToDBus/). *FreeDesktop.org*. Retrieved 22 October 2015.
6. Cocagne, Tom (August 2012). "DBus Overview" (https://pythonhosted.org/txdbus/dbus_overview.html). *pythonhosted.org*. Retrieved 22 October 2015.
7. Vermeulen, Jeroen (14 Jul 2013). "Introduction to D-Bus" (http://www.freedesktop.org/wiki/IntroductionToDBus/). *FreeDesktop.org*. Retrieved 3 October 2015. "D-Bus [...] is designed for use as a unified middleware layer underneath the main free desktop environments."
8. Palmieri, John (January 2005). "Get on D-BUS" (https://web.archive.org/web/20151023072022/http://www.redhat.com/magazine/003jan05/features/dbus/). Red Hat Magazine. Archived from the original (https://www.redhat.com/magazine/003jan05/features/dbus/) on 23 October 2015. Retrieved 3 November 2015.
9. "gdbus" (https://developer.gnome.org/gio/stable/gdbus.html). *GNOME developer*. Retrieved 4 January 2015.
10. "QtDBus module" (http://doc.qt.io/qt-5/qtdbus-index.html). *Qt Project*. Retrieved 1 June 2015.
11. "DBus-Java Documentation" (http://dbus.freedesktop.org/doc/dbus-java/). *FreeDesktop.org*. Retrieved 4 January 2015.
12. Poettering, Lennart (19 June 2015). "The new sd-bus API of systemd" (http://0pointer.net/blog/the-new-sd-bus-api-of-systemd.html). Retrieved 21 October 2015.
13. Pennington, Havoc; Wheeler, David; Walters, Colin. "D-Bus Tutorial" (http://dbus.freedesktop.org/doc/dbus-tutorial.html). Retrieved 21 October 2015. "For the within-desktop-session use case, the GNOME and KDE desktops have significant previous experience with different IPC solutions such as CORBA and DCOP. D-Bus is built on that experience and carefully tailored to meet the needs of these desktop projects in particular."
14. Vermeulen, Jeroen (14 Jul 2013). "Introduction to D-Bus" (http://www.freedesktop.org/wiki/IntroductionToDBus/). *FreeDesktop.org*. Retrieved 3 October 2015. " D-Bus was first built to replace the CORBA-like component model underlying the GNOME desktop environment. Similar to DCOP (which is used by KDE), D-Bus is set to become a standard component of the major free desktop environments for GNU/Linux and other platforms."
15. Pennington, Havoc; Carlsson, Anders; Larsson, Alexander; Herzberg, Sven; McVittie, Simon; Zeuthen, David. "D-Bus Specification" (http://dbus.freedesktop.org/doc/dbus-specification.html). *Freedesktop.org*. Retrieved 22 October 2015.
16. Pennington, Havoc; Wheeler, David; Walters, Colin. "D-Bus Tutorial" (http://dbus.freedesktop.org/doc/dbus-tutorial.html). Retrieved 21 October 2015.
17. Poettering, Lennart (19 June 2015). "The new sd-bus API of systemd" (http://0pointer.net/blog/the-new-sd-bus-api-of-systemd.html). Retrieved 21 October 2015. "we are working on moving things to a true user bus, of which there is only one per user on a system, regardless how many times that user happens to log in"

18. Love, Robert (5 January 2005). "Get on the D-BUS" (http://www.linuxjournal.com/article/7744). *Linux Journal*. Retrieved 14 October 2014.
19. "What is D-Bus?" (http://www.freedesktop.org/wiki/Software/dbus/). *FreeDesktop.org*. Retrieved 29 October 2015. "There are also some reimplementations of the D-Bus protocol for languages such as C#, Java, and Ruby. These do not use the libdbus reference implementation"
20. "What is D-Bus?" (http://www.freedesktop.org/wiki/Software/dbus/). *FreeDesktop.org*. Retrieved 29 October 2015. "is built on top of a general one-to-one message passing framework, which can be used by any two apps to communicate directly (without going through the message bus daemon)"
21. "D-BUS System Activation" (https://dbus.freedesktop.org/doc/system-activation.txt). *FreeDesktop.org*. Retrieved 18 February 2016.
22. Palmieri, John (9 Nov 2006). "[announce] D-Bus 1.0.0 "Blue Bird" released" (http://lists.freedesktop.org/archives/dbus/2006-November/006337.html). *dbus* (Mailing list).
23. Molkentin, Daniel (12 November 2006). "D-Bus 1.0 "Blue Bird" Released" (https://dot.kde.org/2006/11/12/d-bus-10-blue-bird-released). *KDE News*. Retrieved 3 November 2015.
24. Seigo, Aaron. "Introduction To D-BUS" (https://techbase.kde.org/Development/Tutorials/D-Bus/Introduction). *KDE TechBase*. Retrieved 3 November 2015.
25. Poettering, Lennart (19 June 2015). "The new sd-bus API of systemd" (http://0pointer.net/blog/the-new-sd-bus-api-of-systemd.html). Retrieved 21 October 2015. "Since systemd's inception it has been the IPC system it exposes its interfaces on."
26. "Polkit reference manual" (http://www.freedesktop.org/software/polkit/docs/latest/polkit.8.html). *FreeDesktop.org*. Retrieved 3 November 2015.
27. "What is D-Bus?" (http://www.freedesktop.org/wiki/Software/dbus/#index1h1). *FreeDesktop.org*. Retrieved 5 January 2015. "The low-level implementation is not primarily designed for application authors to use. Rather, it is a basis for binding authors and a reference for reimplementations. If you are able to do so it is recommended that you use one of the higher level bindings or implementations."
28. "Migrating to GDBus" (https://developer.gnome.org/gio/stable/ch35.html). *GNOME Developer*. Retrieved 21 October 2015. "dbus-glib uses the libdbus reference implementation, GDBus doesn't. Instead, it relies on GIO streams as transport layer, and has its own implementation for the D-Bus connection setup and authentication."
29. "MATE: Roadmap" (http://wiki.mate-desktop.org/roadmap). Retrieved 31 January 2019.
30. Poettering, Lennart (20 Mar 2013). "[HEADSUP] libsystemd-bus + kdbus plans" (http://lists.freedesktop.org/archives/systemd-devel/2013-March/009797.html). *systemd-devel* (Mailing list).
31. Edge, Jake (30 May 2013). "ALS: Linux inter-process communication and kdbus" (https://lwn.net/Articles/551969/). LWN.net. Retrieved 21 October 2015.
32. Poettering, Lennart (19 Jun 2015). "[ANNOUNCE] systemd v221" (http://lists.freedesktop.org/archives/systemd-devel/2015-June/033170.html). *systemd-devel* (Mailing list).
33. "The unveiling of kdbus" (https://lwn.net/Articles/580194/). LWN.net. 2014-01-13.
34. "Documentation/kdbus.txt (from the initial patch set)" (https://lwn.net/Articles/619069/). LWN.net. 2014-11-04.
35. Corbet, Jonathan (13 January 2014). "The unveiling of kdbus" (https://lwn.net/Articles/580194/). LWN.net. Retrieved 11 April 2014.
36. Kroah-Hartman, Greg (13 Apr 2015). "[GIT PULL] kdbus for 4.1-rc1" (http://lkml.iu.edu/hypermail/linux/kernel/1504.1/03936.html). *linux-kernel* (Mailing list).
37. Corbet, Jonathan (22 April 2015). "The kdbuswreck" (https://lwn.net/Articles/641275/). LWN.net. Retrieved 29 June 2015.

38. "Keynote: A Fireside Chat with Greg Kroah-Hartman, Linux Foundation Fellow" (https://www.youtube.com/watch?v=s2I_7uCto5Q&t=20m34s). YouTube. 18 October 2016. Archived (https://ghostarchive.org/varchive/youtube/20211221/s2I_7uCto5Q) from the original on 2021-12-21.
39. "D-Bus Bindings" (http://www.freedesktop.org/wiki/Software/DBusBindings/). *FreeDesktop.org*. Retrieved 5 January 2015.

External links

- D-Bus (http://www.freedesktop.org/Software/dbus) home page at Freedesktop.org
- D-Bus specification (http://dbus.freedesktop.org/doc/dbus-specification.html)
- Introduction to D-Bus (http://www.freedesktop.org/wiki/IntroductionToDBus) on the Freedesktop.org wiki
- D-Bus Tutorial (http://dbus.freedesktop.org/doc/dbus-tutorial.html)
- DBus Overview (https://pythonhosted.org/txdbus/dbus_overview.html)

Retrieved from "https://en.wikipedia.org/w/index.php?title=D-Bus&oldid=1085433458"

This page was last edited on 30 April 2022, at 14:47 (UTC).