NAME

dbus-daemon - Message bus daemon

SYNOPSIS

dbus-daemon

dbus-daemon [--version] [--session] [--system] [--config-file=*FILE*] [--print-address [=*DESCRIPTOR*]] [--print-pid [=*DESCRIPTOR*]] [--fork]

DESCRIPTION

dbus-daemon is the D-Bus message bus daemon. See **http://www.freedesktop.org/software/dbus/** for more information about the big picture. D-Bus is first a library that provides one-to-one communication between any two applications; **dbus-daemon** is an application that uses this library to implement a message bus daemon. Multiple programs connect to the message bus daemon and can exchange messages with one another.

There are two standard message bus instances: the systemwide message bus (installed on many systems as the messagebus init service) and the per-user-login-session message bus (started each time a user logs in). **dbus-daemon** is used for both of these instances, but with a different configuration file.

The --session option is equivalent to --config-file=/etc/dbus-1/session.conf and the --system option is equivalent to --config-file=/etc/dbus-1/system.conf. By creating additional configuration files and using the --config-file option, additional special-purpose message bus daemons could be created.

The systemwide daemon is normally launched by an init script, standardly called simply messagebus.

The systemwide daemon is largely used for broadcasting system events, such as changes to the printer queue, or adding/removing devices.

The per-session daemon is used for various interprocess communication among desktop applications (however, it is not tied to X or the GUI in any way).

SIGHUP will cause the D-Bus daemon to PARTIALLY reload its configuration file and to flush its user/group information caches. Some configuration changes would require kicking all apps off the bus; so they will only take effect if you restart the daemon. Policy changes should take effect with SIGHUP.

OPTIONS

The following options are supported:

--config-file=FILE
Use the given configuration file.

--fork
Force the message bus to fork and become a daemon, even if the configuration file does not specify that it should. In most contexts the configuration file already gets this right, though. This option is not supported on Windows.

--nofork
Force the message bus not to fork and become a daemon, even if the configuration file specifies that it should. On Windows, the dbus-daemon never forks, so this option is allowed but does nothing.

--print-address[=DESCRIPTOR]
Print the address of the message bus to standard output, or to the given file descriptor. This is used by programs that launch the message bus.

--print-pid[=DESCRIPTOR]
Print the process ID of the message bus to standard output, or to the given file descriptor.

This is used by programs that launch the message bus.

--session
Use the standard configuration file for the per-login-session message bus.

--system
Use the standard configuration file for the systemwide message bus.

--version
Print the version of the daemon.

--introspect
Print the introspection information for all D-Bus internal interfaces.

--address[=ADDRESS]
Set the address to listen on. This option overrides the address configured in the configuration file.

--systemd-activation
Enable systemd-style service activation. Only useful in conjunction with the systemd system and session manager on Linux.

--nopidfile
Dont write a PID file even if one is configured in the configuration files.

CONFIGURATION FILE

A message bus daemon has a configuration file that specializes it for a particular application. For example, one configuration file might set up the message bus to be a systemwide message bus, while another might set it up to be a per-user-login-session bus.

The configuration file also establishes resource limits, security parameters, and so forth.

The configuration file is not part of any interoperability specification and its backward compatibility is not guaranteed; this document is documentation, not specification.

The standard systemwide and per-session message bus setups are configured in the files /etc/dbus-1/system.conf and /etc/dbus-1/session.conf. These files normally <include> a system-local.conf or session-local.conf; you can put local overrides in those files to avoid modifying the primary configuration files.

The configuration file is an XML document. It must have the following doctype declaration:

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

The following elements may be present in the configuration file.

- *<busconfig>*

Root element.

- *<type>*

The well-known type of the message bus. Currently known values are system and session; if other values are set, they should be either added to the D-Bus specification, or namespaced. The last <type> element wins (previous values are ignored). This element only controls which message bus specific environment variables are set in activated clients. Most of the policy that distinguishes a session bus from the system bus is controlled from the other elements in the configuration file.

If the well-known type of the message bus is session, then the DBUS_STARTER_BUS_TYPE environment variable will be set to session and the DBUS_SESSION_BUS_ADDRESS environment variable will be set to the address of the session bus. Likewise, if the type of the message bus is system, then the DBUS_STARTER_BUS_TYPE environment variable will be set to system and the DBUS_SESSION_BUS_ADDRESS environment variable will be set to the address of the system bus (which is normally well known anyway).

Example: <type>session</type>

- *<include>*

Include a file <include>filename.conf</include> at this point. If the filename is relative, it is located relative to the configuration file doing the including.

<include> has an optional attribute ignore_missing=(yes|no) which defaults to no if not provided. This attribute controls whether its a fatal error for the included file to be absent.

- *<includedir>*

Include all files in <includedir>foo.d</includedir> at this point. Files in the directory are included in undefined order. Only files ending in .conf are included.

This is intended to allow extension of the system bus by particular packages. For example, if CUPS wants to be able to send out notification of printer queue changes, it could install a file to /etc/dbus-1/system.d that allowed all apps to receive this message and allowed the printer daemon user to send it.

- *<user>*

The user account the daemon should run as, as either a username or a UID. If the daemon cannot change to this UID on startup, it will exit. If this element is not present, the daemon will not change or care about its UID.

The last <user> entry in the file wins, the others are ignored.

The user is changed after the bus has completed initialization. So sockets etc. will be created before changing user, but no data will be read from clients before changing user. This means that sockets and PID files can be created in a location that requires root privileges for writing.

- *<fork>*

If present, the bus daemon becomes a real daemon (forks into the background, etc.). This is generally used rather than the --fork command line option.

- *<keep_umask>*

If present, the bus daemon keeps its original umask when forking. This may be useful to avoid affecting the behavior of child processes.

- *<syslog>*

If present, the bus daemon will log to syslog.

- *<pidfile>*

If present, the bus daemon will write its pid to the specified file. The --nopidfile command-line option takes precedence over this setting.

- *<allow_anonymous>*

If present, connections that authenticated using the ANONYMOUS mechanism will be authorized to connect. This option has no practical effect unless the ANONYMOUS mechanism has also been enabled using the *<auth>* element, described below.

- *<listen>*

Add an address that the bus should listen on. The address is in the standard D-Bus format that contains a transport name plus possible parameters/options.

Example: <listen>unix:path=/tmp/foo</listen>

Example: <listen>tcp:host=localhost,port=1234</listen>

If there are multiple <listen> elements, then the bus listens on multiple addresses. The bus will pass its address to started services or other interested parties with the last address given in <listen> first. That is, apps will try to connect to the last <listen> address first.

tcp sockets can accept IPv4 addresses, IPv6 addresses or hostnames. If a hostname resolves to multiple addresses, the server will bind to all of them. The family=ipv4 or family=ipv6 options can be used to force it to bind to a subset of addresses

Example: <listen>tcp:host=localhost,port=0,family=ipv4</listen>

A special case is using a port number of zero (or omitting the port), which means to choose an available port selected by the operating system. The port number chosen can be obtained with the --print-address command line parameter and will be present in other cases where the server reports its own address, such as when DBUS_SESSION_BUS_ADDRESS is set.

Example: <listen>tcp:host=localhost,port=0</listen>

tcp/nonce-tcp addresses also allow a bind=hostname option, used in a listenable address to configure the interface on which the server will listen: either the hostname is the IP address of one of the local machines interfaces (most commonly 127.0.0.1), or a DNS name that resolves to one of those IP addresses, or * to listen on all interfaces simultaneously. If not specified, the default is the same value as host.

Example: <listen>tcp:host=localhost,bind=*,port=0</listen>

- *<auth>*

Lists permitted authorization mechanisms. If this element doesnt exist, then all known mechanisms are allowed. If there are multiple <auth> elements, all the listed mechanisms are allowed. The order in which mechanisms are listed is not meaningful.

Example: <auth>EXTERNAL</auth>

Example: <auth>DBUS_COOKIE_SHA1</auth>

- *<servicedir>*

Adds a directory to scan for .service files. Directories are scanned starting with the first to appear in the config file (the first .service file found that provides a particular service will be used).

Service files tell the bus how to automatically start a program. They are primarily used with the per-user-session bus, not the systemwide bus.

- *<standard_session_servicedirs/>*

<standard_session_servicedirs/> is equivalent to specifying a series of <servicedir/> elements for each of the data directories in the XDG Base Directory Specification with the subdirectory dbus-1/services, so for example /usr/share/dbus-1/services would be among the directories searched.

The XDG Base Directory Specification can be found at **http://freedesktop.org/wiki/Standards/basedir-spec** if it hasnt moved, otherwise try your favorite search engine.

The <standard_session_servicedirs/> option is only relevant to the per-user-session bus daemon defined in /etc/dbus-1/session.conf. Putting it in any other configuration file would probably be nonsense.

- *<standard_system_servicedirs/>*

<standard_system_servicedirs/> specifies the standard system-wide activation directories that should be searched for service files. This option defaults to /usr/share/dbus-1/system-services.

The <standard_system_servicedirs/> option is only relevant to the per-system bus daemon defined in /etc/dbus-1/system.conf. Putting it in any other configuration file would probably be nonsense.

- *<servicehelper/>*

<servicehelper/> specifies the setuid helper that is used to launch system daemons with an alternate user. Typically this should be the dbus-daemon-launch-helper executable in located in libexec.

The <servicehelper/> option is only relevant to the per-system bus daemon defined in /etc/dbus-1/system.conf. Putting it in any other configuration file would probably be nonsense.

- *<limit>*

<limit> establishes a resource limit. For example:

<limit name=max_message_size>64</limit>
<limit name=max_completed_connections>512</limit>

The name attribute is mandatory. Available limit names are:

max_incoming_bytes : total size in bytes of messages
incoming from a single connection
max_incoming_unix_fds : total number of unix fds of messages
incoming from a single connection
max_outgoing_bytes : total size in bytes of messages
queued up for a single connection
max_outgoing_unix_fds : total number of unix fds of messages
queued up for a single connection
max_message_size : max size of a single message in
bytes
max_message_unix_fds : max unix fds of a single message
service_start_timeout : milliseconds (thousandths) until
a started service has to connect
auth_timeout : milliseconds (thousandths) a
connection is given to
authenticate
pending_fd_timeout : milliseconds (thousandths) a
fd is given to be transmitted to
dbus-daemon before disconnecting the
connection
max_completed_connections : max number of authenticated connections
max_incomplete_connections : max number of unauthenticated
connections
max_connections_per_user : max number of completed connections from
the same user
max_pending_service_starts : max number of service launches in
progress at the same time
max_names_per_connection : max number of names a single
connection can own
max_match_rules_per_connection: max number of match rules for a single
connection
max_replies_per_connection : max number of pending method
replies per connection
(number of calls-in-progress)
reply_timeout : milliseconds (thousandths)
until a method call times out

The max incoming/outgoing queue sizes allow a new message to be queued if one byte remains below the max. So you can in fact exceed the max by max_message_size.

max_completed_connections divided by max_connections_per_user is the number of users that can work together to denial-of-service all other users by using up all connections on the systemwide bus.

Limits are normally only of interest on the systemwide bus, not the user session buses.

- *<policy>*

The <policy> element defines a security policy to be applied to a particular set of connections to the bus. A policy is made up of <allow> and <deny> elements. Policies are normally used with the systemwide bus; they are analogous to a firewall in that they allow expected traffic and prevent unexpected traffic.

Currently, the system bus has a default-deny policy for sending method calls and owning bus names. Everything else, in particular reply messages, receive checks, and signals has a default allow policy.

In general, it is best to keep system services as small, targeted programs which run in their own

process and provide a single bus name. Then, all that is needed is an <allow> rule for the own permission to let the process claim the bus name, and a send_destination rule to allow traffic from some or all uids to your service.

The <policy> element has one of four attributes:

context=(default|mandatory)
at_console=(true|false)
user=username or userid
group=group name or gid

Policies are applied to a connection as follows:

- all context=default policies are applied
- all group=connections users group policies are applied
in undefined order
- all user=connections auth user policies are applied
in undefined order
- all at_console=true policies are applied
- all at_console=false policies are applied
- all context=mandatory policies are applied

Policies applied later will override those applied earlier, when the policies overlap. Multiple policies with the same user/group/context are applied in the order they appear in the config file.

<*deny*>
    <*allow*>

A <deny> element appears below a <policy> element and prohibits some action. The <allow> element makes an exception to previous <deny> statements, and works just like <deny> but with the inverse meaning.

The possible attributes of these elements are:

send_interface=interface_name
send_member=method_or_signal_name
send_error=error_name
send_destination=name
send_type=method_call | method_return | signal | error
send_path=/path/name

receive_interface=interface_name
receive_member=method_or_signal_name
receive_error=error_name
receive_sender=name
receive_type=method_call | method_return | signal | error
receive_path=/path/name

send_requested_reply=true | false
receive_requested_reply=true | false

eavesdrop=true | false

own=name
own_prefix=name
user=username
group=groupname

Examples:

<deny send_destination=org.freedesktop.Service send_interface=org.freedesktop.System send_member=Reboot/
<deny send_destination=org.freedesktop.System/>
<deny receive_sender=org.freedesktop.System/>

<deny user=john/>
<deny group=enemies/>

The <deny> elements attributes determine whether the deny matches a particular action. If it matches, the action is denied (unless later rules in the config file allow it).

send_destination and receive_sender rules mean that messages may not be sent to or received from the *owner* of the given name, not that they may not be sent *to that name*. That is, if a connection owns services A, B, C, and sending to A is denied, sending to B or C will not work either.

The other send_* and receive_* attributes are purely textual/by-value matches against the given field in the message header.

Eavesdropping occurs when an application receives a message that was explicitly addressed to a name the application does not own, or is a reply to such a message. Eavesdropping thus only applies to messages that are addressed to services and replies to such messages (i.e. it does not apply to signals).

For <allow>, eavesdrop=true indicates that the rule matches even when eavesdropping. eavesdrop=false is the default and means that the rule only allows messages to go to their specified recipient. For <deny>, eavesdrop=true indicates that the rule matches only when eavesdropping. eavesdrop=false is the default for <deny> also, but here it means that the rule applies always, even when not eavesdropping. The eavesdrop attribute can only be combined with send and receive rules (with send_* and receive_* attributes).

The [send|receive]_requested_reply attribute works similarly to the eavesdrop attribute. It controls whether the <deny> or <allow> matches a reply that is expected (corresponds to a previous method call message). This attribute only makes sense for reply messages (errors and method returns), and is ignored for other message types.

For <allow>, [send|receive]_requested_reply=true is the default and indicates that only requested replies are allowed by the rule. [send|receive]_requested_reply=false means that the rule allows any reply even if unexpected.

For <deny>, [send|receive]_requested_reply=false is the default but indicates that the rule matches only when the reply was not requested. [send|receive]_requested_reply=true indicates that the rule applies always, regardless of pending reply state.

user and group denials mean that the given user or group may not connect to the message bus.

For name, username, groupname, etc. the character * can be substituted, meaning any. Complex globs like foo.bar.* arent allowed for now because theyd be work to implement and maybe encourage sloppy security anyway.

<allow own_prefix=a.b/> allows you to own the name a.b or any name whose first dot-separated elements are a.b: in particular, you can own a.b.c or a.b.c.d, but not a.bc or a.c. This is useful when services like Telepathy and ReserveDevice define a meaning for subtrees of well-known names, such as org.freedesktop.Telepathy.ConnectionManager.(anything) and org.freedesktop.ReserveDevice1.(anything).

It does not make sense to deny a user or group inside a <policy> for a user or group; user/group denials can only be inside context=default or context=mandatory policies.

A single <deny> rule may specify combinations of attributes such as send_destination and send_interface and send_type. In this case, the denial applies only if both attributes match the message being denied. e.g. <deny send_interface=foo.bar send_destination=foo.blah/> would deny messages with the given interface AND the given bus name. To get an OR effect you specify multiple <deny> rules.

You cant include both send_ and receive_ attributes on the same rule, since whether the message can be sent and whether it can be received are evaluated separately.

Be careful with send_interface/receive_interface, because the interface field in messages is optional. In particular, do NOT specify <deny send_interface=org.foo.Bar/>! This will cause no-interface messages to be blocked for all services, which is almost certainly not what you intended. Always use rules of the form: <deny send_interface=org.foo.Bar send_destination=org.foo.Service/>

- *<selinux>*

The <selinux> element contains settings related to Security Enhanced Linux. More details below.

- *<associate>*

An <associate> element appears below an <selinux> element and creates a mapping. Right now only one kind of association is possible:

<associate own=org.freedesktop.Foobar context=foo_t/>

This means that if a connection asks to own the name org.freedesktop.Foobar then the source context will be the context of the connection and the target context will be foo_t - see the short discussion of SELinux below.

Note, the context here is the target context when requesting a name, NOT the context of the connection owning the name.

Theres currently no way to set a default for owning any name, if we add this syntax it will look like:

<associate own=* context=foo_t/>

If you find a reason this is useful, let the developers know. Right now the default will be the security context of the bus itself.

If two <associate> elements specify the same name, the element appearing later in the configuration file will be used.

SELINUX

See **http://www.nsa.gov/selinux/** for full details on SELinux. Some useful excerpts:

Every subject (process) and object (e.g. file, socket, IPC object, etc) in the system is assigned a collection of security attributes, known as a security context. A security context contains all of the security attributes associated with a particular subject or object that are relevant to the security policy.

In order to better encapsulate security contexts and to provide greater efficiency, the policy enforcement code of SELinux typically handles security identifiers (SIDs) rather than security contexts. A SID is an integer that is mapped by the security server to a security context at runtime.

When a security decision is required, the policy enforcement code passes a pair of SIDs (typically the SID of a subject and the SID of an object, but sometimes a pair of subject SIDs or a pair of object SIDs), and an object security class to the security server. The object security class indicates the kind of object, e.g. a process, a regular file, a directory, a TCP socket, etc.

Access decisions specify whether or not a permission is granted for a given pair of SIDs and class. Each object class has a set of associated permissions defined to control operations on objects with that class.

D-Bus performs SELinux security checks in two places.

First, any time a message is routed from one connection to another connection, the bus daemon will check permissions with the security context of the first connection as source, security context of the second connection as target, object class dbus and requested permission send_msg.

If a security context is not available for a connection (impossible when using UNIX domain sockets), then the target context used is the context of the bus daemon itself. There is currently no way to change this default, because were assuming that only UNIX domain sockets will be used to connect to the systemwide bus. If this changes, well probably add a way to set the default

connection context.

Second, any time a connection asks to own a name, the bus daemon will check permissions with the security context of the connection as source, the security context specified for the name in the config file as target, object class dbus and requested permission acquire_svc.

The security context for a bus name is specified with the <associate> element described earlier in this document. If a name has no security context associated in the configuration file, the security context of the bus daemon itself will be used.

DEBUGGING

If youre trying to figure out where your messages are going or why you arent getting messages, there are several things you can try.

Remember that the system bus is heavily locked down and if you havent installed a security policy file to allow your message through, it wont work. For the session bus, this is not a concern.

The simplest way to figure out whats happening on the bus is to run the *dbus-monitor* program, which comes with the D-Bus package. You can also send test messages with *dbus-send*. These programs have their own man pages.

If you want to know what the daemon itself is doing, you might consider running a separate copy of the daemon to test against. This will allow you to put the daemon under a debugger, or run it with verbose output, without messing up your real session and system daemons.

To run a separate test copy of the daemon, for example you might open a terminal and type:

DBUS_VERBOSE=1 dbus-daemon --session --print-address

The test daemon address will be printed when the daemon starts. You will need to copy-and-paste this address and use it as the value of the DBUS_SESSION_BUS_ADDRESS environment variable when you launch the applications you want to test. This will cause those applications to connect to your test bus instead of the DBUS_SESSION_BUS_ADDRESS of your real session bus.

DBUS_VERBOSE=1 will have NO EFFECT unless your copy of D-Bus was compiled with verbose mode enabled. This is not recommended in production builds due to performance impact. You may need to rebuild D-Bus if your copy was not built with debugging in mind. (DBUS_VERBOSE also affects the D-Bus library and thus applications using D-Bus; it may be useful to see verbose output on both the client side and from the daemon.)

If you want to get fancy, you can create a custom bus configuration for your test bus (see the session.conf and system.conf files that define the two default configurations for example). This would allow you to specify a different directory for .service files, for example.

AUTHOR

See **http://www.freedesktop.org/software/dbus/doc/AUTHORS**

BUGS

Please send bug reports to the D-Bus mailing list or bug tracker, see **http://www.freedesktop.org/software/dbus/**