systemd.socket

- Description
- Automatic Dependencies
  - Implicit Dependencies
  - Default Dependencies
- Options
- See Also

systemd

systemd.socket

5

systemd.socket

Socket unit configuration

`socket.socket`

Description

A unit configuration file whose name ends in `.socket` encodes information about an IPC or network socket or a file system FIFO controlled and supervised by systemd, for socket-based activation.

This man page lists the configuration options specific to this unit type. See `systemd.unit(5)` for the common options of all unit configuration files. The common configuration items are configured in the generic [Unit] and [Install] sections. The socket specific configuration options are configured in the [Socket] section.

Additional options are listed in `systemd.exec(5)`, which define the execution environment the `ExecStartPre=`, `ExecStartPost=`, `ExecStopPre=` and `ExecStopPost=` commands are executed in, and in `systemd.kill(5)`, which define the way the processes are terminated, and in `systemd.resource-control(5)`, which configure resource control settings for the processes of the socket.

For each socket unit, a matching service unit must exist, describing the service to start on incoming traffic on the socket (see `systemd.service(5)` for more information about .service units). The name of the .service unit is by default the same as the name of the .socket unit, but can be altered with the `Service=` option described below. Depending on the setting of the `Accept=` option described below, this .service unit must either be named like the .socket unit, but with the suffix replaced, unless overridden with `Service=`; or it must be a template unit named the same way. Example: a socket file `foo.socket` needs a matching service `foo.service` if `Accept=no` is set. If `Accept=yes` is set, a service template `foo@.service` must exist from which services are instantiated for each incoming connection.

No implicit `WantedBy=` or `RequiredBy=` dependency from the socket to the service is added. This means that the service may be started without the socket, in which case it must be able to open sockets by itself. To prevent this, an explicit `Requires=` dependency may be added.

Socket units may be used to implement on-demand starting of services, as well as parallelized starting of services. See the blog stories linked at the end for an introduction.

Note that the daemon software configured for socket activation with socket units needs to be able to accept sockets from systemd, either via systemd's native socket passing interface (see `sd_listen_fds(3)` for details about the precise protocol used and the order in which the file descriptors are passed) or via traditional `inetd(8)`-style socket passing (i.e. sockets passed in via standard input and output, using `StandardInput=socket` in the service file).

By default, network sockets allocated through `.socket` units are allocated in the host's network namespace (see `network_namespaces(7)`). This does not mean however that the service activated by a configured socket unit has to be part of the host's network namespace as well. It is supported and even good practice to run services in their own network namespace (for example through `PrivateNetwork=`, see `systemd.exec(5)`), receiving only the sockets configured through socket-activation from the host's namespace. In such a set-up communication within the host's network namespace is only permitted through the activation sockets passed in while all sockets allocated from the service code itself will be associated with the service's own namespace, and thus possibly subject to a restrictive configuration.

Alternatively, it is possible to run a `.socket` unit in another network namespace by setting `PrivateNetwork=yes` in combination with `JoinsNamespaceOf=`, see `systemd.exec(5)` and `systemd.unit(5)` for details.

Automatic Dependencies

Implicit Dependencies

The following dependencies are implicitly added:

- Socket units automatically gain a `Before=` dependency on the service units they activate.
- Socket units referring to file system paths (such as `AF_UNIX` sockets or FIFOs) implicitly gain `Requires=` and `After=` dependencies on all mount units necessary to access those paths.
- Socket units using the `BindToDevice=` setting automatically gain a `BindsTo=` and `After=` dependency on the device unit encapsulating the specified network interface.

Additional implicit dependencies may be added as result of execution and resource control parameters as documented in `systemd.exec(5)` and `systemd.resource-control(5)`.

Default Dependencies

The following dependencies are added unless `DefaultDependencies=no` is set:

- Socket units automatically gain a `Before=` dependency on `sockets.target`.
- Socket units automatically gain a pair of `After=` and `Requires=` dependency on `sysinit.target`, and a pair of `Before=` and `Conflicts=` dependencies on `shutdown.target`. These dependencies ensure that the socket unit is started before normal services at boot, and is stopped on shutdown. Only sockets involved with early boot or late system shutdown should disable `DefaultDependencies=` option.

Options

Socket unit files may include [Unit] and [Install] sections, which are described in `systemd.unit(5)`.

Socket unit files must include a [Socket] section, which carries information about the socket or FIFO it supervises. A number of options that may be used in this section are shared with other unit types. These options are documented in `systemd.exec(5)`, `systemd.kill(5)` and `systemd.resource-control(5)`. The options specific to the [Socket] section of socket units are the following:

`ListenStream=`; `ListenDatagram=`; `ListenSequentialPacket=`
Specifies an address to listen on for a stream (`SOCK_STREAM`), datagram (`SOCK_DGRAM`), or sequential packet (`SOCK_SEQPACKET`) socket, respectively. The address can be written in various formats:

If the address starts with a slash (`/`), it is read as file system socket in the `AF_UNIX` socket family.

If the address starts with an at symbol (`@`), it is read as abstract namespace socket in the `AF_UNIX` family. The `@` is replaced with a `NUL` character before binding. For details, see `unix(7)`.

If the address string is a single number, it is read as port number to listen on via IPv6. Depending on the value of `BindIPv6Only=` (see below) this might result in the service being available via both IPv6 and IPv4 (default) or just via IPv6.

If the address string is a string in the format `v.w.x.y:z`, it is interpreted as IPv4 address <v.w.x.y> and port <z>.

If the address string is a string in the format `[x]:y`, it is interpreted as IPv6 address <x> and port <y>. An optional interface scope (interface name or number) may be specified after a `%` symbol: `[x]:y%dev`. Interface scopes are only useful with link-local addresses, because the kernel ignores them in other cases. Note that if an address is specified as IPv6, it might still make the service available via IPv4 too, depending on the `BindIPv6Only=` setting (see below).

If the address string is a string in the format `vsock:x:y`, it is read as CID <x> on a port <y> address in the `AF_VSOCK` family. The CID is a unique 32-bit integer identifier in `AF_VSOCK` analogous to an IP address. Specifying the CID is optional, and may be set to the empty string. `vsock` may be replaced with `vsock-stream`, `vsock-dgram` or `vsock-seqpacket` to force usage of the corresponding socket type.

Note that `SOCK_SEQPACKET` (i.e. `ListenSequentialPacket=`) is only available for `AF_UNIX` sockets. `SOCK_STREAM` (i.e. `ListenStream=`) when used for IP sockets refers to TCP sockets, `SOCK_DGRAM` (i.e. `ListenDatagram=`) to UDP.

These options may be specified more than once, in which case incoming traffic on any of the sockets will trigger service activation, and all listed sockets will be passed to the service, regardless of whether there is incoming traffic on them or not. If the empty string is assigned to any of these options, the list of addresses to listen on is reset, all prior uses of any of these options will have no effect.

It is also possible to have more than one socket unit for the same service when using `Service=`, and the service will receive all the sockets configured in all the socket units. Sockets configured in one unit are passed in the order of configuration, but no ordering between socket units is specified.

If an IP address is used here, it is often desirable to listen on it before the interface it is configured on is up and running, and even regardless of whether it will be up and running at any point. To deal with this, it is recommended to set the `FreeBind=` option described below.

`ListenFIFO=`
Specifies a file system FIFO (see `fifo(7)` for details) to listen on. This expects an absolute file system path as argument. Behavior otherwise is very similar to the `ListenDatagram=` directive above.

`ListenSpecial=`
Specifies a special file in the file system to listen on. This expects an absolute file system path as argument. Behavior otherwise is very similar to the `ListenFIFO=` directive above. Use this to open character device nodes as well as special files in `/proc/` and `/sys/`.

`ListenNetlink=`
Specifies a Netlink family to create a socket for to listen on. This expects a short string referring to the `AF_NETLINK` family name (such as `audit` or `kobject-uevent`) as argument, optionally suffixed by a whitespace followed by a multicast group integer. Behavior otherwise is very similar to the `ListenDatagram=` directive above.

`ListenMessageQueue=`
Specifies a POSIX message queue name to listen on (see `mq_overview(7)` for details). This expects a valid message queue name (i.e. beginning with `/`). Behavior otherwise is very similar to the `ListenFIFO=` directive above. On Linux message queue descriptors are actually file descriptors and can be inherited between processes.

`ListenUSBFunction=`
Specifies a USB FunctionFS endpoints location to listen on, for implementation of USB gadget functions. This expects an absolute file system path of a FunctionFS mount point as the argument. Behavior otherwise is very similar to the `ListenFIFO=` directive above. Use this to open the FunctionFS endpoint `ep0`. When using this option, the activated service has to have the `USBFunctionDescriptors=` and `USBFunctionStrings=` options set.

`SocketProtocol=`
Takes one of `udplite`, `sctp` or `mptcp`. The socket will use the UDP-Lite (`IPPROTO_UDPLITE`), SCTP (`IPPROTO_SCTP`) or MPTCP (`IPPROTO_MPTCP`) protocol, respectively.

`BindIPv6Only=`
Takes one of `default`, `both` or `ipv6-only`. Controls the IPV6_V6ONLY socket option (see `ipv6(7)` for details). If `both`, IPv6 sockets bound will be accessible via both IPv4 and IPv6. If `ipv6-only`, they will be accessible via IPv6 only. If `default` (which is the default, surprise!), the system wide default setting is used, as controlled by `/proc/sys/net/ipv6/bindv6only`, which in turn defaults to the equivalent of `both`.

`Backlog=`
Takes an unsigned 32-bit integer argument. Specifies the number of connections to queue that have not been accepted yet. This setting matters only for stream and sequential packet sockets. See `listen(2)` for details. Defaults to 4294967295. Note that this value is silently capped by the `net.core.somaxconn` sysctl, which typically defaults to 4096, so typically the sysctl is the setting that actually matters.

`BindToDevice=`
Specifies a network interface name to bind this socket to. If set, traffic will only be accepted from the specified network interfaces. This controls the `SO_BINDTODEVICE` socket option (see `socket(7)` for details). If this option is used, an implicit dependency from this socket unit on the network interface device unit is created (see `systemd.device(5)`). Note that setting this parameter might result in additional dependencies to be added to the unit (see above).

`SocketUser=`; `SocketGroup=`
Takes a UNIX user/group name. When specified, all `AF_UNIX` sockets, FIFO nodes, and message

queues are owned by the specified user and group. If unset (the default), the nodes are owned by the root user/group (if run in system context) or the invoking user/group (if run in user context). If only a user is specified but no group, then the group is derived from the user's default group.

`SocketMode=`
If listening on a file system socket, FIFO, or message queue, this option specifies the file system access mode used when creating the file node. Takes an access mode in octal notation. Defaults to 0666.

`DirectoryMode=`
If listening on a file system socket or FIFO, the parent directories are automatically created if needed. This option specifies the file system access mode used when creating these directories. Takes an access mode in octal notation. Defaults to 0755.

`Accept=`
Takes a boolean argument. If yes, a service instance is spawned for each incoming connection and only the connection socket is passed to it. If no, all listening sockets themselves are passed to the started service unit, and only one service unit is spawned for all connections (also see above). This value is ignored for datagram sockets and FIFOs where a single service unit unconditionally handles all incoming traffic. Defaults to `no`.

Typically, for performance sensitive services, a choice of `Accept=no` is preferable, since that way only the first connection will have to pay the activation resource cost. On the other hand, for sporadically used services `Accept=yes` can be preferable as it simplifies the implementation (as the service program code only has to process a single connection instead of handling multiple) and enables stronger security (since the various sandboxing options can be used to isolate parallel connections from each other, as each is serviced by a separate service instance and process).

A service listening on an `AF_UNIX` socket may, but does not need to, call `close(2)` on the received socket before exiting. However, it must not unlink the socket from a file system. It should not invoke `shutdown(2)` on sockets it got with `Accept=no`, but it may do so for sockets it got with `Accept=yes` set.

Setting `Accept=yes` is in particular useful for allowing daemons designed for usage with `inetd(8)` to work unmodified with systemd socket activation.

Note that depending on this setting the services activated by units of this type are either regular services (in case of `Accept=``no`) or instances of templated services (in case of `Accept=``yes`). See the Description section above for a more detailed discussion of the naming rules of triggered services.

For IPv4 and IPv6 connections, the `$REMOTE_ADDR` environment variable will contain the remote IP address, and `$REMOTE_PORT` will contain the remote port number. These two variables correspond to those defined by the CGI interface for web services (see RFC 3875).

For `AF_UNIX` socket connections, the `$REMOTE_ADDR` environment variable will contain either the remote socket's file system path starting with a slash (`/`) or its address in the abstract namespace starting with an at symbol (`@`). If the socket is unnamed, `$REMOTE_ADDR` will not be set.

If `Accept=yes` is used, the activated service process will have set the `$SO_COOKIE` environment variable to the Linux socket cookie, formatted as decimal integer. The socket cookie can otherwise be acquired via `getsockopt(7)`.

It is recommended to set `CollectMode=inactive-or-failed` for service instances activated via `Accept=yes`, to ensure that failed connection services are cleaned up and released from memory, and do not accumulate.

`Writable=`
Takes a boolean argument. May only be used in conjunction with `ListenSpecial=`. If true, the specified special file is opened in read-write mode, if false, in read-only mode. Defaults to false.

`FlushPending=`
Takes a boolean argument. May only be used when `Accept=no`. If yes, the socket's buffers are cleared after the triggered service exited. This causes any pending data to be flushed and any pending incoming connections to be rejected. If no, the socket's buffers will not be cleared, permitting the service to handle any pending connections after restart, which is the usually expected behaviour. Defaults to `no`.

`MaxConnections=`
The maximum number of connections to simultaneously run services instances for, when `Accept=yes` is set. If more concurrent connections are coming in, they will be refused until at least one existing connection is terminated. This setting has no effect on sockets configured with `Accept=no` or datagram sockets. Defaults to 64.

`MaxConnectionsPerSource=`
The maximum number of connections for a service per source IP address (in case of IPv4/IPv6), per source CID (in case of `AF_VSOCK`), or source UID (in case of `AF_UNIX`). This is very similar to the `MaxConnections=` directive above. Defaults to 0, i.e. disabled.

`KeepAlive=`
Takes a boolean argument. If true, the TCP/IP stack will send a keep alive message after 2h (depending on the configuration of `/proc/sys/net/ipv4/tcp_keepalive_time`) for all TCP streams accepted on this socket. This controls the `SO_KEEPALIVE` socket option (see `socket(7)` and the TCP Keepalive HOWTO for details.) Defaults to `false`.

`KeepAliveTimeSec=`
Takes time (in seconds) as argument. The connection needs to remain idle before TCP starts sending keepalive probes. This controls the TCP_KEEPIDLE socket option (see `socket(7)` and the TCP Keepalive HOWTO for details.) Default value is 7200 seconds (2 hours).

`KeepAliveIntervalSec=`
Takes time (in seconds) as argument between individual keepalive probes, if the socket option `SO_KEEPALIVE` has been set on this socket. This controls the `TCP_KEEPINTVL` socket option (see `socket(7)` and the TCP Keepalive HOWTO for details.) Default value is 75 seconds.

`KeepAliveProbes=`
Takes an integer as argument. It is the number of unacknowledged probes to send before considering the connection dead and notifying the application layer. This controls the TCP_KEEPCNT socket option (see `socket(7)` and the TCP Keepalive HOWTO for details.) Default value is 9.

`NoDelay=`
Takes a boolean argument. TCP Nagle's algorithm works by combining a number of small outgoing messages, and sending them all at once. This controls the TCP_NODELAY socket option (see `tcp(7)`). Defaults to `false`.

`Priority=`
Takes an integer argument controlling the priority for all traffic sent from this socket. This controls the `SO_PRIORITY` socket option (see `socket(7)` for details.).

`DeferAcceptSec=`
Takes time (in seconds) as argument. If set, the listening process will be awakened only when data arrives on the socket, and not immediately when connection is established. When this option is set, the `TCP_DEFER_ACCEPT` socket option will be used (see `tcp(7)`), and the kernel will ignore initial ACK packets without any data. The argument specifies the approximate amount of time the kernel should wait for incoming data before falling back to the normal behavior of honoring empty ACK packets. This option is beneficial for protocols where the client sends the data first (e.g. HTTP, in contrast to SMTP), because the server process will not be woken up unnecessarily before it can take any action.

If the client also uses the `TCP_DEFER_ACCEPT` option, the latency of the initial connection may be reduced, because the kernel will send data in the final packet establishing the connection (the third packet in the "three-way handshake").

Disabled by default.

`ReceiveBuffer=`; `SendBuffer=`
Takes an integer argument controlling the receive or send buffer sizes of this socket, respectively. This controls the `SO_RCVBUF` and `SO_SNDBUF` socket options (see `socket(7)` for details.). The usual suffixes K, M, G are supported and are understood to the base of 1024.

`IPTOS=`
Takes an integer argument controlling the IP Type-Of-Service field for packets generated from this socket. This controls the `IP_TOS` socket option (see `ip(7)` for details.). Either a numeric string or one of `low-delay`, `throughput`, `reliability` or `low-cost` may be specified.

`IPTTL=`
Takes an integer argument controlling the IPv4 Time-To-Live/IPv6 Hop-Count field for packets generated from this socket. This sets the `IP_TTL`/`IPV6_UNICAST_HOPS` socket options (see `ip(7)` and `ipv6(7)` for details.)

`Mark=`
Takes an integer value. Controls the firewall mark of packets generated by this socket. This can be used in the firewall logic to filter packets from this socket. This sets the `SO_MARK` socket option. See `iptables(8)` for details.

`ReusePort=`
Takes a boolean value. If true, allows multiple `bind(2)`s to this TCP or UDP port. This controls the `SO_REUSEPORT` socket option. See `socket(7)` for details.

`SmackLabel=`; `SmackLabelIPIn=`; `SmackLabelIPOut=`
Takes a string value. Controls the extended attributes `security.SMACK64`, `security.SMACK64IPIN` and `security.SMACK64IPOUT`, respectively, i.e. the security label of the FIFO, or the security label for the incoming or outgoing connections of the socket, respectively. See Smack for details.

`XAttrEntryPoint=`; `XAttrListen=`; `XAttrAccept=`
Set an extended attribute on the socket, in the form `NAME=VALUE`. The extended attribute's name must be in the `user.` namespace, i.e. begin with the four characters `user.`. Specifiers are expanded in both the name and the value, see `systemd.unit(5)` for details on the available specifiers. These settings may be used more than once to set multiple extended attributes; assigning the empty string resets the list.

The three variables control where the extended attributes are applied: `XAttrEntryPoint=` sets them on the file-system inode an `AF_UNIX` socket is bound to (i.e. the socket node visible in the file system at the specified listening path; this only applies to `AF_UNIX` sockets), `XAttrListen=` sets them on the listening socket, and `XAttrAccept=` sets them on the connection sockets accepted off the listening socket (and is thus only relevant in combination with `Accept=``yes`).

This is primarily useful to tag sockets so that they can be discovered and classified by other tools. For example, Varlink entrypoint sockets are supposed to be tagged with `user.varlink=entrypoint` via `XAttrEntryPoint=`, which makes them discoverable via `varlinkctl list-sockets`, see `varlinkctl(1)`.

These settings require kernel support for extended attributes on socket inodes (available since Linux 7.0). Extended attributes that cannot be applied are logged at debug level and otherwise ignored.

`SELinuxContextFromNet=`
Takes a boolean argument. When true, systemd will attempt to figure out the SELinux label used for the instantiated service from the information handed by the peer over the network. Note that only the security level is used from the information provided by the peer. Other parts of the resulting SELinux context originate from either the target binary that is effectively triggered by

socket unit or from the value of the `SELinuxContext=` option. This configuration option applies only when activated service is passed in single socket file descriptor, i.e. service instances that have standard input connected to a socket or services triggered by exactly one socket unit. Also note that this option is useful only when MLS/MCS SELinux policy is deployed. Defaults to `false`.

`PipeSize=`
Takes a size in bytes. Controls the pipe buffer size of FIFOs configured in this socket unit. See `fcntl(2)` for details. The usual suffixes K, M, G are supported and are understood to the base of 1024.

`MessageQueueMaxMessages=`, `MessageQueueMessageSize=`
These two settings control the mq_maxmsg field or the mq_msgsize field, respectively, when creating the message queue. `MessageQueueMaxMessages=` takes an integer value. `MessageQueueMessageSize=` takes a size in bytes, and the usual suffixes K, M, G are supported and understood to the base of 1024. Note that either none or both of these variables need to be set. See `mq_setattr(3)` for details.

`FreeBind=`
Takes a boolean value. Controls whether the socket can be bound to non-local IP addresses. This is useful to configure sockets listening on specific IP addresses before those IP addresses are successfully configured on a network interface. This sets the `IP_FREEBIND`/`IPV6_FREEBIND` socket option. For robustness reasons it is recommended to use this option whenever you bind a socket to a specific IP address. Defaults to `false`.

`Transparent=`
Takes a boolean value. Controls the `IP_TRANSPARENT`/`IPV6_TRANSPARENT` socket option. Defaults to `false`.

`Broadcast=`
Takes a boolean value. This controls the `SO_BROADCAST` socket option, which allows broadcast datagrams to be sent from this socket. Defaults to `false`.

`PassCredentials=`
Takes a boolean value. This controls the `SO_PASSCRED` socket option, which allows `AF_UNIX` sockets to receive the credentials of the sending process in an ancillary message. Defaults to `false`.

`PassPIDFD=`
Takes a boolean value. This controls the `SO_PASSPIDFD` socket option, which allows `AF_UNIX` sockets to receive the pidfd of the sending process in an ancillary message. Defaults to `false`.

`PassSecurity=`
Takes a boolean value. This controls the `SO_PASSSEC` socket option, which allows `AF_UNIX` sockets to receive the security context of the sending process in an ancillary message. Defaults to `false`.

`PassPacketInfo=`
Takes a boolean value. This controls the `IP_PKTINFO`, `IPV6_RECVPKTINFO`, `NETLINK_PKTINFO` or `PACKET_AUXDATA` socket options, which enable reception of additional per-packet metadata as

ancillary message, on `AF_INET`, `AF_INET6`, `AF_UNIX` and `AF_PACKET` sockets. Defaults to `false`.

`AcceptFileDescriptors=`
Takes a boolean value. This controls the `SO_PASSRIGHTS` socket option, which when disabled prohibits the peer from sending `SCM_RIGHTS` ancillary messages (aka file descriptors) via `AF_UNIX` sockets. Defaults to `true`.

`Timestamping=`
Takes one of `off`, `us` (alias: `usec`, `µs`) or `ns` (alias: `nsec`). This controls the `SO_TIMESTAMP` or `SO_TIMESTAMPNS` socket options, and enables whether ingress network traffic shall carry timestamping metadata. Defaults to `off`.

`TCPCongestion=`
Takes a string value. Controls the TCP congestion algorithm used by this socket. Should be one of `westwood`, `reno`, `cubic`, `lp` or any other available algorithm supported by the IP stack. This setting applies only to stream sockets.

`ExecStartPre=`; `ExecStartPost=`
Takes one or more command lines, which are executed before or after the listening sockets/FIFOs are created and bound, respectively. The first token of the command line must be an absolute filename, then followed by arguments for the process. Multiple command lines may be specified following the same scheme as used for `ExecStartPre=` of service unit files.

`ExecStopPre=`; `ExecStopPost=`
Additional commands that are executed before or after the listening sockets/FIFOs are closed and removed, respectively. Multiple command lines may be specified following the same scheme as used for `ExecStartPre=` of service unit files.

`TimeoutSec=`
Configures the time to wait for the commands specified in `ExecStartPre=`, `ExecStartPost=`, `ExecStopPre=` and `ExecStopPost=` to finish. If a command does not exit within the configured time, the socket will be considered failed and be shut down again. All commands still running will be terminated forcibly via `SIGTERM`, and after another delay of this time with `SIGKILL`. (See `KillMode=` in `systemd.kill(5)`.) Takes a unit-less value in seconds, or a time span value such as "5min 20s". Pass `0` to disable the timeout logic. Defaults to `DefaultTimeoutStartSec=` from the manager configuration file (see `systemd-system.conf(5)`).

`Service=`
Specifies the service unit name to activate on incoming traffic. This setting is only allowed for sockets with `Accept=no`. It defaults to the service that bears the same name as the socket (with the suffix replaced). In most cases, it should not be necessary to use this option. Note that setting this parameter might result in additional dependencies to be added to the unit (see above).

`RemoveOnStop=`
Takes a boolean argument. If enabled, any file nodes created by this socket unit are removed when it is stopped. This applies to `AF_UNIX` sockets in the file system, POSIX message queues, FIFOs, as well as any symlinks to them configured with `Symlinks=`. Normally, it should not be necessary to

use this option, and is not recommended as services might continue to run after the socket unit has been terminated and it should still be possible to communicate with them via their file system node. Defaults to off.

`Symlinks=`
Takes a list of file system paths. The specified paths will be created as symlinks to the `AF_UNIX` socket path or FIFO path of this socket unit. If this setting is used, only one `AF_UNIX` socket in the file system or one FIFO may be configured for the socket unit. Use this option to manage one or more symlinked alias names for a socket, binding their lifecycle together. Note that if creation of a symlink fails this is not considered fatal for the socket unit, and the socket unit may still start. If an empty string is assigned, the list of paths is reset. Defaults to an empty list.

`FileDescriptorName=`
Assigns a name to all file descriptors this socket unit encapsulates. This is useful to help activated services identify specific file descriptors, if multiple fds are passed. Services may use the `sd_listen_fds_with_names(3)` call to acquire the names configured for the received file descriptors. Names may contain any ASCII character, but must exclude control characters and `:`, and must be at most 255 characters in length. If this setting is not used, the file descriptor name defaults to the name of the socket unit (including its `.socket` suffix) when `Accept=no`, `connection` otherwise.

`TriggerLimitIntervalSec=`; `TriggerLimitBurst=`
Configures a limit on how often this socket unit may be activated within a specific time interval. The `TriggerLimitIntervalSec=` setting may be used to configure the length of the time interval in the usual time units `us`, `ms`, `s`, `min`, `h`, ... and defaults to 2s (See `systemd.time(7)` for details on the various time units understood). The `TriggerLimitBurst=` setting takes a positive integer value and specifies the number of permitted activations per time interval, and defaults to 200 for `Accept=yes` sockets (thus by default permitting 200 activations per 2s), and 20 otherwise (20 activations per 2s). Set either to 0 to disable any form of trigger rate limiting.

If the limit is hit, the socket unit is placed into a failure mode, and will not be connectible anymore until restarted. Note that this limit is enforced before the service activation is enqueued.

Compare with `PollLimitIntervalSec=`/`PollLimitBurst=` described below, which implements a temporary slowdown if a socket unit is flooded with incoming traffic, as opposed to the permanent failure state `TriggerLimitIntervalSec=`/`TriggerLimitBurst=` results in.

`PollLimitIntervalSec=`; `PollLimitBurst=`
Configures a limit on how often polling events on the file descriptors backing this socket unit will be considered. This pair of settings is similar to `TriggerLimitIntervalSec=`/`TriggerLimitBurst=` but instead of putting a (fatal) limit on the activation frequency puts a (transient) limit on the polling frequency. The expected parameter syntax and range are identical to that of the aforementioned options, and can be disabled the same way.

If the polling limit is hit polling is temporarily disabled on it until the specified time window passes. The polling limit hence slows down connection attempts if hit, but unlike the trigger limit will not cause permanent failures. It's the recommended mechanism to deal with DoS attempts through

packet flooding.

The polling limit is enforced per file descriptor to listen on, as opposed to the trigger limit which is enforced for the entire socket unit. This distinction matters for socket units that listen on multiple file descriptors (i.e. have multiple `ListenXYZ=` stanzas).

These setting defaults to 150 (in case of `Accept=yes`) and 15 (otherwise) polling events per 2s. This is considerably lower than the default values for the trigger limit (see above) and means that the polling limit should typically ensure the trigger limit is never hit, unless one of them is reconfigured or disabled.

`DeferTrigger=`
Takes a boolean argument, or `patient`. May only be used when `Accept=no`. If enabled, job mode `lenient` instead of `replace` is used when triggering the service, which means currently activating/running units that conflict with the service won't be disturbed/brought down. Furthermore, if a conflict exists, the socket unit will wait for current job queue to complete and potentially defer the activation by then. An upper limit of total time to wait can be configured via `DeferTriggerMaxSec=`. If set to `yes`, the socket unit will fail if all jobs have finished or the timeout has been reached but the conflict remains. If `patient`, always wait until `DeferTriggerMaxSec=` elapses. Defaults to no.

This setting is particularly useful if the socket unit should stay active across switch-root/soft-reboot operations while the triggered service is stopped.

`DeferTriggerMaxSec=`
Configures the maximum time to defer the triggering when `DeferTrigger=` is enabled. If the service cannot be activated within the specified time, the socket will be considered failed and get terminated. Takes a unit-less value in seconds, or a time span value such as "5min 20s". Pass `0` or `infinity` to disable the timeout logic (the default).

`PassFileDescriptorsToExec=`
Takes a boolean argument. Defaults to off. If enabled, file descriptors created by the socket unit are passed to `ExecStartPost=`, `ExecStopPre=`, and `ExecStopPost=` commands from the socket unit. The passed file descriptors can be accessed with `sd_listen_fds(3)` as if the commands were invoked from the associated service units. Note that `ExecStartPre=` command cannot access socket file descriptors.

See Also

`systemd(1)`, `systemctl(1)`, `systemd-system.conf(5)`, `systemd.unit(5)`, `systemd.exec(5)`, `systemd.kill(5)`, `systemd.resource-control(5)`, `systemd.service(5)`, `systemd.directives(7)`, `sd_listen_fds(3)`, `sd_listen_fds_with_names(3)`

For more extensive descriptions see the "systemd for Developers" series: Socket Activation, Socket Activation, part II, Converting inetd Services, Socket Activated Internet Services and OS Containers.

□□□ #2
□ □□□ □□□ 2026-10-31 15:07:22 UTC
□ □□□ □□□ 2026-10-31 15:07:23 UTC