NAME

if_bridge - network bridge device

SYNOPSIS

To compile this driver into the kernel, place the following line in your kernel configuration file:

```
device if_bridge
```

Alternatively, to load the driver as a module at boot time, place the following lines in loader.conf(5):

```
if_bridge_load="YES"
bridgestp_load="YES"
```

DESCRIPTION

The **if_bridge** driver creates a logical link between two or more IEEE 802 networks that use the same (or "similar enough") framing format. For example, it is possible to bridge Ethernet and 802.11 networks together, but it is not possible to bridge Ethernet and Token Ring together.

Each **if_bridge** interface is created at runtime using interface cloning. This is most easily done with the ifconfig(8) **create** command or using the *cloned_interfaces* variable in rc.conf(5).

The **if_bridge** interface randomly chooses a link (MAC) address in the range reserved for locally administered addresses when it is created. This address is guaranteed to be unique *only* across all **if_bridge** interfaces on the local machine. Thus you can theoretically have two bridges on different machines with the same link addresses. The address can be changed by assigning the desired link address using ifconfig(8).

If sysctl(8) node *net.link.bridge.inherit_mac* has a non-zero value, the newly created bridge will inherit the MAC address from its first member instead of choosing a random link-level address. This will provide more predictable bridge MAC addresses without any additional configuration, but currently this feature is known to break some L2 protocols, for example PPPoE that is provided by ng_pppoe(4) and ppp(8). Currently this feature is considered as experimental and is turned off by default.

A bridge can be used to provide several services, such as a simple 802.11-to-Ethernet bridge for wireless hosts, or traffic isolation.

A bridge works like a switch, forwarding traffic from one interface to another. Multicast and broadcast packets are always forwarded to all interfaces that are part of the bridge. For unicast traffic, the bridge learns which MAC addresses are associated with which interfaces and will forward the traffic selectively.

By default the bridge logs MAC address port flapping to syslog(3). This behavior can be disabled by setting the sysctl(8) variable *net.link.bridge.log_mac_flap* to 0.

All the bridged member interfaces need to be up in order to pass network traffic. These can be enabled using ifconfig(8) or *ifconfig_<interface>*="up" in rc.conf(5).

The MTU of the first member interface to be added is used as the bridge MTU. All additional members will have their MTU changed to match. If the MTU of a bridge is changed after its creation, the MTU of all member interfaces is also changed to match.

The TOE, TSO, TXCSUM and TXCSUM6 capabilities on all interfaces added to the bridge are disabled if any of the interfaces do not support/enable them. The LRO capability is always disabled. All the capabilities are restored when the interface is removed from the bridge. Changing capabilities at run-time may cause NIC reinit and a link flap.

The bridge supports "monitor mode", where the packets are discarded after bpf(4) processing, and are not processed or forwarded further. This can be used to multiplex the input of two or more interfaces into a single bpf(4) stream. This is useful for reconstructing the traffic for network taps that transmit the RX/TX signals out through two separate interfaces.

IPV6 SUPPORT

if_bridge supports the AF_INET6 address family on bridge interfaces. The following rc.conf(5) variable configures an IPv6 link-local address on bridge0 interface:

```
ifconfig_bridge0_ipv6="up"
```

or in a more explicit manner:

```
ifconfig_bridge0_ipv6="inet6 auto_linklocal"
```

However, the AF_INET6 address family has a concept of scope zone. Bridging multiple interfaces changes the zone configuration because multiple links are merged to each other and form a new single link while the member interfaces still work individually. This means each member interface still has a separate link-local scope zone and the **if_bridge** interface has another single, aggregated link-local scope zone at the same time. This situation is clearly against the description "zones of the same scope cannot overlap" in Section 5, RFC 4007. Although it works in most cases, it can cause some counterintuitive or undesirable behavior in some edge cases when both, the **if_bridge** interface and one of the member interfaces, have an IPv6 address and applications use both of them.

To prevent this situation, **if_bridge** checks whether a link-local scoped IPv6 address is configured on a

member interface to be added and the **if_bridge** interface. When the **if_bridge** interface has IPv6 addresses, IPv6 addresses on the member interface will be automatically removed before the interface is added.

This behavior can be disabled by setting sysctl(8) variable *net.link.bridge.allow_llz_overlap* to 1.

Note that ACCEPT_RTADV and AUTO_LINKLOCAL interface flags are not enabled by default on **if_bridge** interfaces even when *net.inet6.ip6.accept_rtadv* and/or *net.inet6.ip6.auto_linklocal* is set to 1.

SPANNING TREE

The **if_bridge** driver implements the Rapid Spanning Tree Protocol (RSTP or 802.1w) with backwards compatibility with the legacy Spanning Tree Protocol (STP). Spanning Tree is used to detect and remove loops in a network topology.

RSTP provides faster spanning tree convergence than legacy STP, the protocol will exchange information with neighbouring switches to quickly transition to forwarding without creating loops.

The code will default to RSTP mode but will downgrade any port connected to a legacy STP network so is fully backward compatible. A bridge can be forced to operate in STP mode without rapid state transitions via the *proto* command in ifconfig(8).

The bridge can log STP port changes to syslog(3) by setting the *net.link.bridge.log_stp* node using sysctl(8).

PACKET FILTERING

Packet filtering can be used with any firewall package that hooks in via the pfil(9) framework. When filtering is enabled, bridged packets will pass through the filter inbound on the originating interface, on the bridge interface and outbound on the appropriate interfaces. Either stage can be disabled. The filtering behavior can be controlled using sysctl(8):

net.link.bridge.pfil_onlyip Controls the handling of non-IP packets which are not passed to pfil(9). Set to 1 to only allow IP packets to pass (subject to firewall rules), set to 0 to unconditionally pass all non-IP Ethernet frames.

net.link.bridge.pfil_member
Set to 1 to enable filtering on the incoming and outgoing member interfaces, set to 0 to disable it.

net.link.bridge.pfil_bridge Set to 1 to enable filtering on the bridge interface, set to 0 to disable it.

net.link.bridge.pfil_local_phys
: Set to 1 to additionally filter on the physical interface for locally destined packets. Set to 0 to disable this feature.

net.link.bridge.ipfw
: Set to 1 to enable layer2 filtering with ipfirewall(4), set to 0 to disable it. This needs to be enabled for dummynet(4) support. When *ipfw* is enabled, *pfil_bridge* and *pfil_member* will be disabled so that IPFW is not run twice; these can be re-enabled if desired.

net.link.bridge.ipfw_arp
: Set to 1 to enable layer2 ARP filtering with ipfirewall(4), set to 0 to disable it. Requires *ipfw* to be enabled.

ARP and REVARP packets are forwarded without being filtered and others that are not IP nor IPv6 packets are not forwarded when *pfil_onlyip* is enabled. IPFW can filter Ethernet types using **mac-type** so all packets are passed to the filter for processing.

The packets originating from the bridging host will be seen by the filter on the interface that is looked up in the routing table.

The packets destined to the bridging host will be seen by the filter on the interface with the MAC address equal to the packet's destination MAC. There are situations when some of the bridge members are sharing the same MAC address (for example the vlan(4) interfaces: they are currently sharing the MAC address of the parent physical interface). It is not possible to distinguish between these interfaces using their MAC address, excluding the case when the packet's destination MAC address is equal to the MAC address of the interface on which the packet was entered to the system. In this case the filter will see the incoming packet on this interface. In all other cases the interface seen by the packet filter is chosen from the list of bridge members with the same MAC address and the result strongly depends on the member addition sequence and the actual implementation of **if_bridge**. It is not recommended to rely on the order chosen by the current **if_bridge** implementation since it may change in the future.

The previous paragraph is best illustrated with the following pictures. Let

- the MAC address of the incoming packet's destination is **nn:nn:nn:nn:nn:nn**,
- the interface on which packet entered the system is **ifX**,
- **ifX** MAC address is **xx:xx:xx:xx:xx:xx**,
- there are possibly other bridge members with the same MAC address **xx:xx:xx:xx:xx:xx**,

- the bridge has more than one interface that are sharing the same MAC address **yy:yy:yy:yy:yy:yy**; we will call them **vlanY1**, **vlanY2**, etc.

If the MAC address **nn:nn:nn:nn:nn:nn** is equal to **xx:xx:xx:xx:xx:xx** the filter will see the packet on interface **ifX** no matter if there are any other bridge members carrying the same MAC address. But if the MAC address **nn:nn:nn:nn:nn:nn** is equal to **yy:yy:yy:yy:yy:yy** then the interface that will be seen by the filter is one of the **vlanYn**. It is not possible to predict the name of the actual interface without the knowledge of the system state and the **if_bridge** implementation details.

This problem arises for any bridge members that are sharing the same MAC address, not only to the vlan(4) ones: they were taken just as an example of such a situation. So if one wants to filter the locally destined packets based on their interface name, one should be aware of this implication. The described situation will appear at least on the filtering bridges that are doing IP-forwarding; in some of such cases it is better to assign the IP address only to the **if_bridge** interface and not to the bridge members. Enabling *net.link.bridge.pfil_local_phys* will let you do the additional filtering on the physical interface.

NETMAP

netmap(4) applications may open a bridge interface in emulated mode. The netmap application will receive all packets which arrive from member interfaces. In particular, packets which would otherwise be forwarded to another member interface will be received by the netmap application.

When the netmap(4) application transmits a packet to the host stack via the bridge interface, **if_bridge** receive it and attempts to determine its ‘source’ interface by looking up the source MAC address in the interface’s learning tables. Packets for which no matching source interface is found are dropped and the input error counter is incremented. If a matching source interface is found, **if_bridge** treats the packet as though it was received from the corresponding interface and handles it normally without passing the packet back to netmap(4).

EXAMPLES

The following when placed in the file */etc/rc.conf* will cause a bridge called "bridge0" to be created, and will add the interfaces "wlan0" and "fxp0" to the bridge, and then enable packet forwarding. Such a configuration could be used to implement a simple 802.11-to-Ethernet bridge (assuming the 802.11 interface is in ad-hoc mode).

```
cloned_interfaces="bridge0"
ifconfig_bridge0="addm wlan0 addm fxp0 up"
```

For the bridge to forward packets, all member interfaces and the bridge need to be up. The above example would also require:

```
create_args_wlan0="wlanmode hostap"
ifconfig_wlan0="up ssid my_ap mode 11g"
ifconfig_fxp0="up"
```

Consider a system with two 4-port Ethernet boards. The following will cause a bridge consisting of all 8 ports with Rapid Spanning Tree enabled to be created:

```
ifconfig bridge0 create
ifconfig bridge0 \
   addm fxp0 stp fxp0 \
   addm fxp1 stp fxp1 \
   addm fxp2 stp fxp2 \
   addm fxp3 stp fxp3 \
   addm fxp4 stp fxp4 \
   addm fxp5 stp fxp5 \
   addm fxp6 stp fxp6 \
   addm fxp7 stp fxp7 \
   up
```

The bridge can be used as a regular host interface at the same time as bridging between its member ports. In this example, the bridge connects em0 and em1, and will receive its IP address through DHCP:

```
cloned_interfaces="bridge0"
ifconfig_bridge0="addm em0 addm em1 DHCP"
ifconfig_em0="up"
ifconfig_em1="up"
```

The bridge can tunnel Ethernet across an IP internet using the EtherIP protocol. This can be combined with ipsec(4) to provide an encrypted connection. Create a gif(4) interface and set the local and remote IP addresses for the tunnel, these are reversed on the remote bridge.

```
ifconfig gif0 create
ifconfig gif0 tunnel 1.2.3.4 5.6.7.8 up
ifconfig bridge0 create
ifconfig bridge0 addm fxp0 addm gif0 up
```

SEE ALSO

gif(4), ipf(4), ipfw(4), netmap(4), pf(4), ifconfig(8)

HISTORY

The **if_bridge** driver first appeared in FreeBSD 6.0.

AUTHORS

The **bridge** driver was originally written by Jason L. Wright <*jason@thought.net*> as part of an undergraduate independent study at the University of North Carolina at Greensboro.

This version of the **if_bridge** driver has been heavily modified from the original version by Jason R. Thorpe <*thorpej@wasabisystems.com*>.

Rapid Spanning Tree Protocol (RSTP) support was added by Andrew Thompson <*thompsa@FreeBSD.org*>.

BUGS

The **if_bridge** driver currently supports only Ethernet and Ethernet-like (e.g., 802.11) network devices, which can be configured with the same MTU size as the bridge device.