OpenSolaris Bridging

James Carlson
james.d.carlson@sun.com

Version 0.2

Abstract

This document describes layer two bridging, Spanning Tree, and a software design that supports these features in OpenSolaris. [1]

1 Background

Bridging is a general layer two (L2 or datalink) technology that is used to connect together separate L2 subnetworks, allowing communication between attached nodes as if only a single subnetwork were in use. It is generally independent of layer three (L3 or network) technologies, such as IP, although common bridge implementations have special features to enhance operation with popular L3 protocols.

This document covers only Ethernet bridging as described in IEEE 802.1D-1998[1]. It is possible to bridge other types of L2 technologies, such as PPP, and it's also possible to bridge L2 technologies over each other and over other types of transport and even over internetworks, such as with GRE over UDP tunnels. This document does not cover those other possibilities, but does not prohibit future projects from addressing them.

The following subsections provide additional background in the related standards and general bridge design requirements and issues. The next major section describes the implementation of these features in detail.

1.1 Bridging

To construct a bridge between subnetworks, it is necessary to forward packets from one link to another, allowing nodes on those separate networks to communicate. Because L2 technologies (such as Ethernet) typically do not provide protection against forwarding loops, any accidental loops that may occur are tragic: the packets just circle forever at full line rate, rendering the attached subnetworks unusable. If multiple links are involved, the packets may even be amplified.

Because of this issue, prevention of loops is crucial, whether intentionally created for redundancy or accidentally introduced due to miswiring. In 802.1D, the mechanism that prevents loops is the Spanning Tree Protocol (STP). The design for OpenSolaris (described in detail later in this document) includes features in addition to STP to guarantee this safety.

In general, the loop prevention mechanism works by disabling links[2] that, if enabled, would form a forwarding loop in the network. Spanning Tree accomplishes this by computing a loop-free directed graph representing the network; all links that are part of the graph are enabled (made "active"), and those not in the graph are disabled ("inactive"). If the topology changes because of link failure or administrative activity, this graph is computed anew.

In addition to loop prevention, the other important feature of a bridge is the operation of the forwarding mechanism itself[3]. Unlike the network layer, where routing protocols are typically used to distribute information about the locations of resources on the network, the datalink layers typically have no such feature. Instead, destinations within the network are "learned" based on the data traffic seen. If a source address is seen on one link, this means that either this node is attached directly to that link, or it is attached via some other bridge through that link, and that packets

[2]The standards documents refer to MAC layer entities variously as "ports" and "links," and some other sources may call them "interfaces" or "NICs." In this document, I use "link" for consistency with dladm(1M). It should be understood that bridge links may be ordinary devices or aggregations, but not VLANs, tunnels, or VNICs.

[3]The forwarding database is referred to as the "Filtering Database" in 802.1D, and the forwarding of frames to specific links is referred to as "Basic Filtering Services," which emphasizes the fact that this forwarding service is merely an optimization. This design document and the code refer to the process as "forwarding" rather than filtering in order to be consistent with other protocols, and to reduce confusion with L2 Filtering (security-related) services.

destined to that node must be transmitted via that link.

To accomplish this learning function, an Ethernet bridge must listen in promiscuous mode on every link attached to the bridge. Every packet received (regardless of destination MAC address or VLAN) is first inspected for its source address. This inspection tells the bridge the direction in which to send packets for the source address that was seen: when we later receive a packet with this address as a destination, we must send that new packet out via the link over which we originally saw it used as a source address.

Packets that arrive over a link with a destination address pointing back over the same link (as determined by a forwarding look-up) represent "local" traffic. These are not forwarded, but they are still inspected for the MAC source address as part of the learning process.

When we receive a packet where the destination address is one we've never seen before, we must treat it as though it were broadcast, and flood it to every active link except the one over which we received it. The reason for this is that we don't yet know the location of that end node, and it could be anywhere. There are no "subnet routes" involved that would give us the right path, even in an approximate sense, so we must try everywhere. The flooding procedure used in this case is "safe" even though it sends on all links, because Spanning Tree guarantees that the bridged network in use is loop-free, so there are no two paths from any given node that can reach the same node, nor any places where the path would pass through a single node twice.

Note that we do not actually learn the exact location (or connected subnetwork) of the remote node when we see a source address. All that we learn is the correct direction towards that node: there may be other bridges in the path between our system and the end node with that address. Thus, all of the bridges within the bridged subnetwork over time learn which of their links sends a packet towards a given node, and hop-by-hop bridging based on destination address is possible.

Note also that there may be destinations that we simply never learn about. If there are "passive nodes" on the network that never send any packets, and there are other nodes that send unicast packets to them, then we will be forced to treat all of those packets as broadcast, and will waste network bandwidth replicating them along irrelevant paths. This doesn't happen on normal networks, as even "receive only" TCP/IP nodes need to send an occasional ARP message, which reveals L2 location, but it's

possible for this situation to occur on contrived networks where static ARP entries are used, and one-way unicast messages are sent, or where non-IP protocols with statically configured L2 addresses are present. This is an inherent issue with all bridges.

A local link-down notification tells us that all nodes on a given link are no longer reachable, and that forwarding entries through that link should be flushed, but there are no L2 mechanisms to detect a single node (which may be multiple bridge or repeater hops away) going away, experiencing link-down, or changing its attachment point. Thus, it must be possible to fall back to broadcasting to locate a given node in order to cover these cases. This is done by aging away the forwarding entries over time, unless they are seen used as source addresses periodically. When an entry is eventually deleted, we again treat it as an unknown destination, and flood the packet throughout the network. This aging is not synchronized in the network, which means that some nodes may still know the correct path to use, and others may not. Synchronization is not needed because the forwarding mechanism itself is merely an optimization, and flooding always works.

Over time, on a steady-state network, each bridge keeps in its forwarding database only the frequently used destinations that pass through that node. This allows for some amount of scaling, provided that there are at least some conversations that do not pass through a common choke point.

1.2 Multicast

The IEEE 802.1D "Extended Filtering Services" feature can be used to optimize the handling of multicast packets within a bridged network.

For non-IP protocols, 802.1D describes protocols called "Generic Attribute Registration Protocol" (GARP) and "GARP Multicast Registration Protocol" (GMRP). These protocols (particularly GMRP) have the distinct disadvantage of not working especially well in practice. GMRP defines three operational modes for a bridged link: "forward all," "forward unregistered," and "filter unregistered." The only one that's fully interoperable with GMRP-unaware nodes is "forward all" (no filtering), and all normal hosts are GMRP-unaware, meaning that the feature has limited utility.

Thus, for maximum compatibility, we will not implement GARP or GMRP, and will simply replicate all non-IP multicast packets (except for

the well-known 01:80:c2:00:00:0x range used for STP, RSTP, MSTP, and Pause frames; those are never forwarded).

For IP, there are a set of well-known multicast addresses that must be replicated everywhere, and for the other multicast addresses, IGMP and MLD both provide useful clues. IP host nodes announce their interest in seeing particular multicast group traffic by sending an IGMP or MLD Report message specifying the group to be received. IP multicast routers announce their presence on the subnet by sending IGMP or MLD Query messages. Hosts need to see just the specific groups they've joined, while routers must see all multicast messages.

It would thus be possible to avoid forwarding a multicast packet unless either (a) an IGMP or MLD Report message specifically requested that multicast address or (b) any IGMP or MLD Query was seen. This project does not address this issue in the initial phase, but may in the future.

1.3 VLANs

1.3.1 Basic Handling

Virtual LANs (VLANs), and the way in which they're used in bridging, are described in 802.1Q[2]. They are logically located "above" the actions of a bridge. In other words, the use of bridging is an attribute of a link, while multiple VLANs can be configured to run on a given link.

For this reason, it generally does not make sense to talk about attaching a bridge to a VLAN, or attempting to bridge between VLANs. Instead, the bridged network is global, and we talk about the ways in which the VLANs are interconnected among the bridges.

On a given link, a typical bridge will have an "allowed VLAN" set and a single "default VLAN," which is defined by the MAC layer. The allowed VLANs are the tagged packets that will be allowed through: the link is a "member" of these VLANs. The default VLAN is the VLAN ID associated with untagged packets. If an untagged packet is received, it is treated as having the indicated default VLAN, and thus will become tagged when forwarded over another that has a different default VLAN, and all packets with that default VLAN ID that are relayed from other links are sent as untagged.

Solaris is currently missing the required "default VLAN" MAC feature. It erroneously treats untagged packets as being a member of no defined

VLAN at all. This oversight must be corrected by this project.

The CFI bit in the VLAN header is used (along with the E-RIF header) to handle special cases involved with non-Ethernet address translations. These features (CFI and E-RIF) will not be supported.

1.3.2 World According To GARP

802.1Q[2] section 5.3(i) "requires" the use of GARP in order to support "GARP VLAN Registration Protocol" (GVRP).

However, a close reading of section 11 of the standard shows that it is not in fact needed on all bridges, because administrators can restrict the use of GVRP so that it doesn't function. It also poses security problems (in that GVRP has no security at all, and allows remote systems to reconfigure VLAN usage), and has little deployment. For this reason, we will not support GVRP.

A future project could add this support. The required functionality would likely be a user-space daemon that uses the libdladm interfaces to create and destroy VLAN links on demand by the protocol.

1.3.3 IVL versus SVL

It's possible to outfit each VLAN with its own MAC forwarding database, and learn addresses separately on each VLAN. This scheme is called "Independent VLAN Learning" (IVL). It's also possible to use a shared MAC forwarding database (SVL) among VLANs.

The subtle issues with the two approaches are detailed in Annex B of 802.1Q. In short, if you have asymmetric VLANs (where transmit and receive on a given link uses different VLAN IDs with untagged frames), then you need SVL. If you're concerned about the security of VLANs, and need to prevent one VLAN user (such as an exclusive stack instance zone) from causing harm by transmitting packets with the same source MAC address as some user on a separate VLAN, then only IVL will do.

For our purposes, the security issue is far more acute, so we need to have something like IVL support. However, we can go one better by stepping outside the narrow bounds described in the standard. We will have a single database, but with both VLAN and MAC address as keys.

When we learn a new MAC address, we will install an SVL entry in the database, marked with the original VLAN plus a flag to indicate that

VLAN comparisons always match. If we learn the same MAC address on a separate VLAN and a separate link, then we'll clear out that flag, and do IVL (VLAN-specific) matches. This should give us the benefits of both schemes.

In order to support aging properly with this scheme, we will need to detect when all of the remaining entries refer to a single output link for a given MAC address. This check will be triggered when we age away an IVL entry. If we do, and the remainder are on a single link, then all but one are deleted, and that last one is marked SVL again.

The remaining risk is that a passive node (one that never transmits any broadcast [ARP] or multicast [NDP] messages, and never sends a packet to any previously-unknown unicast address) can be locked out by a sender on a different VLAN using the same MAC source address. This seems like an acceptably unlikely situation that we'll choose this mode by default, but may implement an undocumented flag to allow pure IVL mode in case some user needs it.

1.4 Frame Check Sequence

An ordinary 802.1D Ethernet bridge – one that does not deal in VLANs – does not modify the packets in transit, and thus has no need to change the FCS (CRC-32) value. This is a good property, as it allows the communicating end stations to detect any errors that may occur inside intermediate bridges during DMA or local storage of the message along the path. It preserves end-to-end protection.

Even those bridges that deal in VLANs can use the Galois properties of CRCs to compute a modified FCS based on the original value, rather than recomputing from scratch, and thus preserve the end-to-end protection of the FCS.

The best implementation is to check the FCS on reception, but use the same value or one computed based on adjustments to the input value (avoiding regeneration) on transmit.

Unfortunately, the Ethernet drivers in OpenSolaris and the overall system architecture do not permit the client applications to receive the FCS value on input or specify it on output, so the requirement specified in section 6.3.7 of 802.1D will not be met. This could be met with a future project by modifying the MAC layer and the drivers, but the design described in

this document does not address the issue.

More generally, it would be wise to move the check value (such as datalink FCS and network and higher layer checksums) generation and checking as close to actual data use as possible, in order to cover the widest possible range of transmission errors, both internal and external, rather than progressively moving it further away in order to eke out small bits of performance. However, the trade-offs and design criteria for that are outside the scope of this project.

1.5 Spanning Tree

The Spanning Tree Protocol (STP) is used to prevent loops from forming among interconnected bridges. It does this by computing a spanning tree (an acyclic graph that covers all the nodes), and then disabling all of the extra links not used in that tree. By definition, those extra links must each form a loop if used.

It consists of two logical components. The first is a set of Bridge PDUs (BPDUs) that communicate topology information among the bridges. The second is a set of per-link controls and states that enable a given link to forward traffic.

STP and RSTP (the "rapid" variant) do not respect VLANs. As a result, administrators must be careful not to create situations in which there are redundant links that carry different sets of VLANs. All but one of the redundant links will be cut by the actions of STP, and this will lead to isolated VLAN segments. This is an inherent part of STP.

A variant known as "Multiple Spanning Tree Protocol" (MSTP) is capable of handling separate trees for separate VLANs, but only up to a point. This protocol defines up to 64 separate "instances" to which each of the 4094 possible VLANs must be assigned administratively. Each instance is used to build a separate tree, and thus has the same management issues if there are subsets of the VLANs used in a particular MSTP instance carried on a particular link.

MSTP is highly complex, both in implementation and in actual use. It has seen little deployment, and although we plan to investigate implementation of the protocol with this project, we will not plan to include it in the first release. (Fortunately, the administrative controls, though hard to understand, are simple in design; the user merely needs a means to select

the desired abstract MSTP instance number for each VLAN.)

One of the important things to understand about STP is that it "fails open." If a bridge is unable to communicate with other bridges (for example, due to an unfortunate choice of layer two MAC filters that cause BPDUs to be dropped), it will determine that the link has not been connected to another bridge, and thus will enable forwarding. If another bridge is actually present – but unseen by STP – then the result is likely to be a forwarding loop, and network failure. This is an unavoidable consequence of STP design, although constraints on L2 filter specifications (such that they cannot drop BPDUs) would be wise.

1.6 Other Layers

Ethernet bridging occupies a particular niche in the stack. Below the layer at which bridging is performed are link aggregation, link security, and the basic IEEE MAC functions. Above the bridging layer are the VLAN components and the datalink consumers, including virtual links (VNICs) and network layer protocols.

We currently implement link aggregation, so our bridging implementation runs over aggregations, if present. If we implemented the authenticator portion of 802.1X[3], then the "Authorized" link state described in that document would be implemented below aggregation, closer to the actual MAC.

The interfaces above bridging include the internal representation of VLANs as distinct datalink objects, and the network layer protocols and DLPI consumers.

2 Implementation

The major components of the implementation include configuration parameter storage, a user-space control daemon, a doors-based control and status interface, and kernel components that support the data paths. In addition to these major parts, we have several minor features, including an observability node used for snoop/wireshark access, and SMF interaction.

The high level issues regarding SMF integration and the dladm subsystem are covered in detail in the architectural documentation, available

in "Solaris Bridging" (PSARC 2008/055). It is assumed that the reader is familiar with the intended architecture.

2.1 libdladm

2.1.1 Control/Status Interface

Every bridge instance has a running bridge daemon, and every daemon has a door mounted as a file under the `/var/run/bridge_door/` directory, using the bridge instance name as the file name.

This interface gives processes outside of the bridge daemon control over the bridge and access to bridge status. Since control over the daemon (e.g., adding and removing links) is equivalent to rewiring the network, we enforce the same `PRIV_SYS_DL_CONFIG` privilege for the caller as dladm would need for the kernel when write-type operations are performed.

The door commands are:

bdcBridgeGetConfig (unprivileged) returns the Spanning Tree configuration using the `librstp` `UID_STP_CFG_T` structure.

bdcBridgeSetConfig (privileged) sets STP with `UID_STP_CFG_T`.

bdcBridgeGetState (unprivileged) returns Spanning Tree state information using the `librstp` `UID_STP_STATE_T` structure.

bdcBridgeGetPorts (unprivileged) returns a list of `datalink_id_t` values that represent the bridge links.

bdcPortGetConfig (unprivileged) given a port name, returns STP port configuration using the `librstp` structure `UID_STP_PORT_CFG_T`.

bdcPortSetConfig (privileged) sets Spanning Tree port configuration using port name and `UID_STP_PORT_CFG_T`.

bdcPortGetState (unprivileged) given a port name, returns the Spanning Tree port state using the `librstp` `UID_STP_PORT_STATE_T` structure.

bdcPortGetForwarding (unprivileged) given a port name, returns the forwarding enabled or disabled status as a boolean.

bdcPortSetForwarding (privileged) sets forwarding enabled or disabled status using port name and a boolean flag.

bdcPortSetPVID (privileged) sets the default VLAN ID (PVID) using port name and a VLAN tag number.

bdcPortVlanCtl (privileged) adds or removes an allowed VLAN using port name and an internal `bridge_vlan_ctl_t` structure.

bdcPortAdd (privileged) adds a new port to the bridge by name.

bdcPortRemove (privileged) removes an existing port from the bridge by name.

When STP is disabled, the forwarding controls directly change the state of the port. Otherwise, they send a signal to the STP implementation to change port state.

When a port is removed from a bridge through the door interface, the internal state structure for the port is retained in the daemon. This allows us to guarantee that if the port is later added back to the bridge, it will get the same index number, which reduces confusion.

2.1.2 Configuration Parameters

The `dladm_bridge_get_properties` function reads the stored bridge-wide properties. This fetches the configuration information stored in the SMF instance, and is used by `dladm` and `bridged`.

The `dladm_bridge_run_properties` function reads the bridge properties from the daemon, using the doors `bdcBridgeGetConfig` interface.

The `dladm_bridge_configure` and `dladm_bridge_enable` functions create the SMF data and link IDs used for bridge instances. The configure function creates a bridge (if asked), and sets or changes the parameters (priority, max age, hello time, forward delay, and force protocol) associated using the standard `libscf` interfaces. If the bridge is already running, then it also calls the door interface in order to update the parameters on the running instance. When creating a bridge, the enable function is called after any initial links are added, and this sets the bridge running via SMF. Note that the configure function is used both for "create-bridge" and "modify-bridge." The API intentionally does not follow the command line interface exactly, but instead follows the functionality.

The `dladm_bridge_delete` function tears down a running bridge via SMF, removes the SMF configuration, and deletes the link ID used for the bridge.

The `dladm_valid_bridgename` function verifies that a given bridge name can be used by checking that it's a legal link name without a trailing unit number, and that it's less than `MAXLINKNAMELEN` in length. This is used in various places to sanity-check a user-provided bridge name. The `dladm_observe_to_bridge` function converts an observability node name into a bridge name by removing the final "0" character.

The `dladm_bridge_setlink` and `dladm_bridge_getlink` functions assign a link to a bridge, and get the bridge assignment for a link. When a link is successfully assigned to or removed from a bridge, the daemon is notified so that it can make corresponding changes.

A `dladm_bridge_get_portlist` function returns a list of datalink IDs representing the links attached to a given running bridge, and the list is freed by `dladm_bridge_free_portlist`. These functions are used for status reporting in dladm.

Status reporting also uses `dladm_bridge_state`, which invokes bdcBridgeGetState to get the internal bridge state and, for individual links, `dladm_bridge_link_state`, which uses bdcPortGetState.

The new per-link bridging parameters are handled within the dladm library as link properties, named "stp", "stp_p2p", "stp_cost", "stp_edge", "stp_priority", "forward", and "default_tag",

The `dladm_get_linkprop_values` function retrieves these parameters individually for each link (based on `datalink_id_t`). The functions `dladm_bridge_set_port_cfg` and `dladm_bridge_get_port_cfg` access the running values for all but "forward" and "default_tag". These are manipulated with separate functions because they are not part of the librstp interface.

Finally, the library has an internal `dladm_bridge_vlanctl` function that is used by the VLAN logic within libdladm to enable and disable allowed VLANs on a given link as the user creates and destroys VLAN instances.

2.2 Spanning Tree

A brief investigation showed that we can obtain a usable STP and RSTP implementation from the rstplib project on sourceforge. This code hasn't been touched in over 6 years, and hasn't been ported to Solaris or OpenSolaris, but, after doing some simple porting and testing, it appears to be functional.

We will need to keep it as a separate dynamically linked library (named `librstp`), as the source code is under an LGPL license, and direct linking could invite legal problems.

We will also need to make some non-trivial changes to the library. The main problem with it is that the `stp_to.h` functions (`STP_OUT_*`) are all defined as explicit named entry points into the application. The application that links with this library must itself provide functions that conform to the library's expectations. It's a call-back mechanism, but without registration.

This is an unusual design for a library, and is difficult to manage. Instead, we will create a function pointer array inside librstp (`stp_vectors`), and have the calling application pass a pointer to these callback functions via `STP_IN_init`.

A few other minor changes are necessary in order to make the code lint clean and compile correctly on OpenSolaris. These changes have been sent upstream to the maintainer, though, given the age of the library, no further action is expected.

2.3 Kernel Bridging

Early in the prototyping process, we received the demonstration bridging code that had been ported in from BSD by the "ethbridge" project, and enhanced by the Xen/xVM team. We spent some time evaluating the design of the code (which was at that time STREAMS-based) and the fit for this project.

After some discussion, we discovered that the portions that were reusable were in fact trivial, and the bulk of the material was inapplicable. We thus abandoned that software and started over from scratch.

The kernel components for bridging consist of three main parts. The observability node and the MAC layer interfaces are described in subsequent sections. This section deals with instance allocation, deallocation,

locks, and reference counting.

The `bridge_stream_t`, which represents a stream opened by a bridge daemon on the `/dev/bridgectl` control node, is the first data structure created during bridge operation. It is allocated when the stream is opened by the daemon and deallocated on close(9E) (when the daemon exits).

The main data structure is the `bridge_inst_t`, which represents a running bridge. This structure is allocated when the bridged daemon opens the `/dev/bridgectl` control node and issues the `BRIOC_NEWBRIDGE` ioctl. It is freed when the daemon closes this control stream; typically when the daemon exits. Note that the bridge creation ioctl can be issued only once per control stream; each instance has a separate control stream.

This part of the design is part of an important safety issue: we must not allow the kernel to continue to bridge between links when the daemon is not running. The risk is that we might miss a topology change, and end up creating a forwarding loop in the network. For this reason, closure of the control node is destructive. It marks the instance as dead, deletes the resources, and waits for all threads to exit before returning.

The instance structures are in a kept in a linked list using `sys/list.h` and protected by a read/write lock. The expectation is that the number of bridges in a real system will be fairly small, and the frequency of create/delete operations on the bridge instances themselves will be low. If this turns out later not to be true, a more complex table could be used.

The instances are reference-counted. They're allocated by `inst_alloc` and then freed by `inst_free`. References are managed by atomic operations on the `bi_refs` member, and the `bridge_unref` function is used to centralize the reference-drop logic. In order to make the shutdown process stable, it's important that no more references are taken when the `BIF_SHUTDOWN` flag is set, that all reference-taking paths are readers on the list rwlock, and that this flag is set before waiting for reference counts to drop.

In the idle case, one reference to the instance is held by the global linked list, and another is held by the control stream. The reference that's held by the global list is dropped after `BIF_SHUTDOWN` is set (meaning that it can no longer be found by name and thus no more references are possible), and the control stream reference is dropped when close(9E) is called (which typically happens at the time the daemon exits).

The second data structure is the `bridge_link_t`, which contains information and MAC-layer handles for a single link on a bridge. This structure

is allocated by BRIOC_ADDLINK on the control stream, and deallocated by BRIOC_REMLINK. These ioctls are sent by the bridge daemon as a part of normal start-up, and during reconfiguration requests.

Because the /dev/bridgectl device is a STREAMS node, and process context does not necessarily exist during ioctl processing, the driver uses a taskq to defer these operations. The task must call mac_open_by_linkid to get a handle on the device, then set up transmit, notification, and receive handles, and finally call mac_bridge_set and mac_start to enable the link. The link is also placed into MAC_DEVPROMISC mode, so that we see all received packets, and the locally configured unicast addresses are retrieved. (The MAC Layer subsection below has more information about the MAC interface for bridging.)

Links are kept in a simple linked list, attached to a given bridge instance. For the same reason as bridges, this simple structure may be revisited if dictated by future requirements for very large numbers of links on a bridge.

Reference drops are done by link_unref, which spawns a taskq when the last reference drops. Each link holds a reference on the bridge, which is dropped only when link_free completes. This ensures that the instance structure is always available when the MAC layer callbacks occur, because all of the handles are closed before that reference is dropped. It also ensures that the forwarding entries (described below) are all removed before the bridge containing them can be removed.

Finally, there are bridge_fwd_t data structures used to hold forwarding entries. These entries are allocated by the fwd_alloc function, which is called by the bridge learning function (as part of the MAC layer receive and transmit callbacks), and by the intercepted MAC layer unicast address functions. For local unicast functions, we set the BFF_LOCALADDR flag to indicate that the address is "special" – it's one that's configured locally and thus shouldn't be overridden by anything learned.

These are kept in an AVL tree attached to the bridge instance, and the reference counts are managed by fwd_find, which returns an entry with the count incremented, and fwd_unref, which drops a reference. The forwarding entries include a list of output links, and hold references to each output link. (As a special case, any forwarding entries that have zero outputs would hold a reference on the bridge itself in order to make the AVL tree stable. Such cases are not currently necessary.)

During forwarding, we hold a reference to the forwarding entry being

used. This guarantees that the output links and bridge cannot be deallocated while those calls are made.

During learning, if a MAC source address is seen arriving on a different link than previously learned, then (as part of the data path) we delete the old forwarding entry and create a new one. There is also a periodic timer that flushes these learned entries. More details about this timer are in the next subsection.

The Crossbow project has a similar structure called `vnic_flow_t`, and referred to as a "classifier." The differences are:

1. The Crossbow tables are per MAC instance (per link), while bridging requires tables that are per bridge.

2. The Crossbow entries are in a simple ad-hoc linked list (not using the common `sys/list.h`), which may be appropriate for a very small number of VNICs configured on a NIC, but will not work acceptably well for hundreds or thousands of learned forwarding entries.

3. The Crossbow entry identifies a single VNIC for delivery using a callback mechanism, but the bridge forwarding mechanism identifies a list of output links.

4. Failing to find an entry in the Crossbow classifier means that the packet isn't delivered. In bridging, it means that the packet must be delivered to all links in the bridge instance except the input link.

5. The Crossbow classifier is implemented up at the VNIC level, while bridging decisions must occur lower in the stack, right after aggregation completes, and before the NIC stream (used by a VNIC) is even identified.

6. Removal of Crossbow classifier entries requires a blocking wait for threads to exit, because the callers assume that VNICs are associated, while bridge forwarding entries need to be flushed quickly on learning new data.

For reference, in order to make Crossbow classifier entries work for bridging, we would have to do the following to make it function properly:

1. Either replicate all entries on each of the member links (an N-by-M explosion in storage and time) or redesign the classifier so that it stores entries per bridge rather than per link.

2. Redesign the Crossbow search mechanism so that it scales much better; it needs to go above a small handful of entries and still handle both input and output traffic, which is necessary in order to deal with looped back (local delivery) packets.

3. Find a better way to insert and remove entries, so that lengthy blocking operations and synchronization are not needed in the middle of the data path. Note that bridging does all of its learning (including both inserts and deletes on the table) in the data path.

In short, the Crossbow classifier entries are analogous to `ipif_t` entries in IP, while the bridge forwarding entries are similar in character to `ire_t`. One is used for local delivery and IP interface identification, with administrative implications, while the other is used for routing and is ephemeral in nature. For this reason, we will not be reusing this technology.

It may be possible to base Crossbow classification on the bridge forwarding mechanism, as, unlike the classifier, bridge forwarding is at least done early enough in the process that it could identify a particular VNIC. However, given the likely performance issues with bridging (see the performance section later in this document), it seems unwise to do that as well.

2.4 Observability Node

In order to enable some observability into the behavior of the bridge itself (in addition to the member links), we are introducing a `/dev/bridge/` directory, served by the `dev` file system. Nodes in this directory are similar to the ones found in `/dev/net/`, but instead of representing vanity-named links, they represent running bridges. Each bridge instance has a node, which can be used with snoop or wireshark to observe traffic flowing through the bridge. (Note that these nodes are passive only; attempts to plumb any network layer protocol on top will fail. This feature is accomplished by the use of the new `mac_no_active` function, described in the next subsection.)

The observability node is a mac- and dls-based driver. An important design issue with these drivers is that there is no way to organize a tear-down sequence: there's no way to force errors among the clients or arrange for a graceful shut-down. Instead, drivers are expected to poll `dls_devnet_destroy` and `mac_unregister`. If success is returned, then all clients are gone. If not, then the device is still busy, and the instance may not go away.

This poses a problem with the bridge instance logic, which is intentionally designed to tear down the instance when "bridged" terminates. To resolve this, a separate `bridge_mac_t` structure is allocated to represent the observability node. This structure is kept in a separate global `bmac_list` linked list. It is allocated by `bmac_alloc` when a new bridge instance starts up, and is freed by the action of the periodic `bridge_timer` function (which also ages forwarding entries) after the bridge instance has been terminated.

When a new bridge instance starts up, we scan the `bmac_list` to find an existing observability node by that name. If there is one, then we use it. This has the helpful side-effect that a snoop session on the bridge can survive a bridge shutdown and restart.

A subtle design point here is that `dls_devnet_create` may need to be redone during instance start-up if `dls_devnet_destroy` (called by the timer) succeeded, but `mac_unregister` had not succeeded. That state is handled by the `BMF_DLS` flag.

To create the observability nodes under `/dev/bridge/`, a new extension has been added to the `dev` file system driver. This portion is simple and quite similar to the existing `devnet` vnode operations, except that it calls the bridging module in order to enumerate the bridge instances.

One important but obscure high-level design issue discovered during prototyping was that `fs/dev` is force-loaded by `startup_modules`. Because that module is now dependent on `drv/bridge` (to pick up the instance enumeration functions), that means the bridge module is loaded first. But the bridge module depends on `drv/dld` in order to create the observability nodes, and that module calls `taskq_create` early during the `_init` processing, due to a change made by Clearview. That causes a panic, as taskqs aren't initialized that early.

Detangling this chain of dependencies would be difficult, as would removing the force-load or redesigning taskqs to be initialized earlier, but there is a simpler solution that we can adopt: just move the `taskq_create`

call to the attach(9E) function instead.

Finally, the bridge_open entry point detects observability nodes by checking the minor node number: zero is reserved for /dev/bridgectl, and all others are redirected by setting q_qinfo to point to a separate set of vectors referencing DLD.

2.5 MAC Layer

There are two main changes to the MAC layer in support of bridging. The first is the concept of a snoop-only MAC device. This is implemented by a new mac_no_active function for drivers, and a mi_disallow_active flag on the mac_impl_t. When this flag is set, mac_do_active_set returns B_FALSE.

The second change is the layering of VNICs atop bridges, and then bridges atop underlying links. The existing Nevada code already implements VNICs atop underlying links, so the same basic design scheme is used to insert bridges. The elements of this structure are:

1. A bridge instance notifies the MAC layer that a given link is part of a bridge by calling mac_bridge_set. This function saves off a pointer to the bridge transmit function and the "getcapab" function, and sets mi_bridge_present. It then sends MAC_NOTE_BRIDGE to all of the clients so that they can refetch the transmit pointers.

2. The mac_capab_get and mac_vnic_capab_get functions change, and a new mac_bridge_capab_get function is added. The change is to call the interposing bridge "getcapab" function if a bridge is present and either the caller is a VNIC or the caller is a normal client and no VNIC is present.

3. The mac_rx and mac_active_rx functions change, and a new function called mac_bridge_rx is added. The change is to make a normal driver deliver to all clients, a bridge deliver to regular clients and VNICs, and a VNIC deliver only to ordinary clients.

4. The mac_rx_add and mac_active_rx_add functions change, and a new mac_low_rx_add function is added. The change is to remove the existing mrf_active flag and use mrf_layer instead, which is defined as an enumeration (mltNormal, mltVnic, and mltBridge).

This allows the receive functions described previously to distinguish between client types.

5. The `mac_vnic_tx_get` and `mac_tx_get` functions change and a new function `mac_bridge_tx_get` is added. The `mi_vnic_txinfo` handle is returned when a VNIC is present and the caller is normal, `mi_bridge_txinfo` when a bridge is present and the caller is a VNIC or is normal and no VNIC is present, and either `mi_txloopinfo` or `mi_txinfo` in the remaining cases.

On tear-down, there's a race condition induced with the MAC layer. The problem is that while `mac_bridge_clear` could hold the `mi_tx_lock` reader/writer lock (or some other appropriate lock) while clearing out the `mi_bridge_present` flag, guaranteeing safety would mean that the `mac_bridge_tx` function would have to hold that same lock for the entire period while the `mi_bridge_xmit` function is called. That can (and does) call functions outside of the MAC layer and that can reenter, so holding that lock is clearly unacceptable.

Instead, the bridge module provides a pointer to `mac_bridge_set` for a function that can take or drop a reference to a bridge link. This function (stored in `mi_bridge_ref`) is called after `mi_bridge_present` is checked but before dropping the lock or calling the transmit function.

In this manner, we guarantee that once `mac_bridge_clear` returns, there are no more new calls to `mac_bridge_tx` that result in calls through the `mi_bridge_xmit` pointer. There may still be threads executing there, but each of them holds a reference to the link in question, and will drop it when done.

2.6 Daemon

The daemon reads the bridge instance configuration from SMF, sets up the Spanning Tree library (librstp), and then iterates over the link information from libdladm, adding links to the Spanning Tree state machine.

It uses libdlpi to open each link and adds a `pfmod` filter that accepts only STP messages. It then relays STP packets between librstp and libdlpi.

There is one daemon per configured (and enabled) bridge. When the daemon terminates, the kernel will stop forwarding for that bridge instance. It must do so for safety reasons, as Spanning Tree is no longer providing loop protection at that point.

2.6.1 Security

The daemon provides a private door interface that is used to extract live statistics from STP, and add and remove links on the fly. When a write-type command is received, `PRIV_SYS_DL_CONFIG` is checked for the invoker.

Auditing is the responsibility of the components that manage the configuration parameters for bridging. Processes with enough privilege to change configuration without invoking auditing can already do so, and must be trusted.

(This area may have architectural weaknesses due to the lack of auditing for datalink.conf; at the time when this document was written, the issues appear to be general for dladm and should probably be addressed that way.)

2.7 Performance Issues

The use of bridging on a link can impose important performance restrictions on the system as a whole. This subsection describes the issues that are known.

The obvious issue is the use of promiscuous mode. This is a required part of bridging, and it's thus not something we can design around, but it is important to note. In addition to the extra load on the system to handle all packets on the wire (rather than just the ones addressed to the local machine), some device drivers reportedly will switch to a lower performance case (disabling interrupt load balancing) when promiscuous mode is enabled.

The less obvious issue is with negotiated capabilities, such as checksum offload. Because a packet that the TCP/IP stack attempts to send on one link may need to be transmitted on a different link due to the action of bridging, the special capabilities that the stack uses on transmit must be a subset of those available on all of the links attached to the bridge.

This capability issue isn't inherent with bridging. One way to work around it would be to have bridge-active links report all capabilities to the upper level software, and then perform in software the ones that the selected output link does not have. This could even be a MAC-wide design policy, making all of the links look the same to IP with respect to features, and using MAC emulation of the hardware acceleration features where

necessary.

However, given the position of bridging in the OpenSolaris portfolio – it's something expected to be used for dedicated or special purposes, and not likely to be found on primary links for a large web server – we will defer this issue for the first implementation.

3 Related Projects

A future project will extend bridging to handle RBridges, using the IETF's "Transparent Interconnection of Lots of Links" (TRILL).

The Clearview project is creating a non-STREAMS common control node for datalink administration. This same facility should be used for the bridge control node (`/dev/bridgectl` when it's available, rather than using a STREAMS device.

4 Unaddressed Issues And Future Work

TBD

Appendices

A Public Documentation License Notice

References

[1] *Part 3: Media Access Control (MAC) Bridges*, ANSI/IEEE Std 802.1D, 1998 Edition (ISO/IEC 15802-3: 1998).

[2] *Virtual Bridged Local Area Networks*, ANSI/IEEE Std 802.1Q, 2005 Ed.

[3] *Port-Based Network Access Control*, ANSI/IEEE Std 802.1X, 2004 Edition.