OpenSolaris Bridging

James Carlson
james.d.carlson@sun.com

Version 1.3

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.

This document also does not cover the operation of TRILL, although that is an important part of the overall project. A separate document will cover TRILL and will build on the design described here.

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

[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 the expected 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.

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 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 the bridge later receive a packet with this address as a destination, it must send that new packet out via the link over which it originally saw the destination 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 the bridge receives a packet where the destination address is one it has never seen before, it must treat the address as though it were broadcast, and flood it to every active link except the one over which it was received. This is because the bridge doesn't yet know the location of that end node, and it could be anywhere. There are no "subnet routes" involved that would yield the right path, even in an approximate sense, so the system must try all locations. 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. Spanning Tree disables the links that would be unsafe.

The bridge does not actually learn the exact location (or connected subnetwork) of the remote node within the overall network when it sees a source address. Instead, what it learns is the currently correct direction towards that node: there may be other bridges in the path between this bridge 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 the bridge simply never learns 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 the bridges will be forced to treat all of those packets as broadcast, and will thus 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.

Nodes in the network may be shut down or moved. A local link-down notification tells the bridge 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, the bridge will 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 Self-Protection

The learning process (like MAC in general) has no inherent security, and is thus vulnerable to a number of different attacks. The implementation of IVL (as in the previous section) provides VLAN isolation, but within a VLAN, a rogue node could misdirect traffic by sending frames with an-

other's MAC address as the source, or could overwhelm the system by sending a huge number of frames with arbitrary source addresses.

To defend against some known flooding attacks, there will be two limits on the forwarding table. The first is a limit on the rate at which entries are added to the table by any link. Each time the bridge module sees a source address that causes it to delete or add an entry, a per-link counter will be incremented. If the count goes over a configurable per-link limit (default: 1000), then further updates will be inhibited, but traffic will still be forwarded. The counter will be decremented by a configurable per-link amount (default: 200) every scan interval (set to 5 seconds). These two will be represented as link parameters named "learn_limit" and "learn_decay".

The second limit will be a per-bridge limit on the table size. If this size is met, then new entries will not be stored (although forwarding continues to operate, and changes are allowed), and the next aging scan will be more aggressive in removing old entries. This will be stored in SMF.

Other than via "svccfg" and "dladm set-linkprop", which are lower-level configuration interfaces, none of these configurable parameters will be included in the normal administrative commands for bridging, at least in the first release. We do not expect administrators to need to configure these values. If a substantial need arises, then they will be promoted in stability.

Note that table overflow is not a hazard. Removing an entry (for any reason) does not cause a loss of connectivity: the forwarding table is actually a kind of "anti cache" in bridging. When a destination isn't found, the bridge will send to all links, rather than dropping. Thus, having entries lost from the table means that efficiency is lost, but that hosts can always still communicate. (In fact, in terms of defending against attack, the systems that have their addresses aged out of the bridge forwarding tables are at an advantage: their traffic will not be diverted.)

One simple remaining attack, however, is to use another station's MAC address in order to disrupt communication and intercept traffic. Notably, if there's no bridge in your network, then you have no defense against this whatsoever, so bridges cannot make the problem any worse. In theory, a bridge could provide some defense by being more cautious in its learning process (or allowing for static forwarding entries), but the limit of that sort of fix is that on a given link (with repeaters), you still cannot have protection. Defense in this area likely depends on new MAC-layer security protocols and is for further study.

1.3 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, this project will not include 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.

1.4 VLANs

1.4.1 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.

1.4.2 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, this project 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.

1.4.3 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.

1.5 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.

Those Ethernet bridges that do deal in VLANs per 802.1Q must modify frames to add a VLAN tag (if the frame is untagged and the VLAN is not the output port's PVID) or remove one (if the frame is tagged and the VLAN is equal to the output port's PVID), per section 8.1.7. Even those bridges that deal in VLANs, though, 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 for a bridge, therefore, is to check the FCS on reception, and then use the same value or, for VLAN support, a new value computed based on adjustments to the input value (avoiding complete FCS 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.

The OpenSolaris bridging implementation will rely on drivers to receive and check FCS on input, and to regenerate the FCS on output for every frame.

The trade-offs and design criteria for system-wide preservation of FCS are outside the scope of this project.

1.6 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. This would fit logically into the existing STP dladm control scheme.)

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.

1.7 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.

Solaris currently implements link aggregation, so this bridging imple-

mentation runs over aggregations, if present. If Solaris in the future implements 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, VNICs, and the network layer protocols and DLPI consumers.

1.8 Optional Installation

The bridging user-space code will be delivered by `SUNWbridger` (root) and `SUNWbridgeu` (usr). These are optional components and are present by default only in `SUNWCall` and `SUNWCXall`. Because `dladm` is always present, we must make the user interface able to deal with the lack of functionality. When the bridging packages are not installed, the `dladm` bridging-related subcommands will return an appropriate error message.

1.9 Dynamic Reconfiguration

In order to allow for Dynamic Reconfiguration (DR) events, the bridging project will create a new `SUNW_bridge_rcm.so` module to remove a link from its bridge (if any) before removing it from the system, and to update links with cached bridge information during a replacement.

2 Implementation

The major components of the implementation include configuration parameter storage, a user-space control daemon, a door-based status interface, and kernel components that support the data paths. In addition to these major parts, this project has 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 Door-Based 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 provides access to bridge status information. None requires special privilege to access.

The door commands are:

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

bdcBridgeGetState returns STP state data with `UID_STP_STATE_T` from `librstp`.

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

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

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

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

The above door commands are implemented in libdladm as a set of functions to simplify use.

The `dladm_bridge_run_properties` function reads the bridge properties from the daemon using `bdcBridgeGetConfig`. The bridge and STP global state is accessed by the `dladm_bridge_state` function, which uses `bdcBridgeGetState`.

A `dladm_bridge_get_portlist` function returns a list of datalink IDs for the links on a given running bridge using `bdcBridgeGetPorts`, and the list is freed by `dladm_bridge_free_portlist`.

Per-link state information is returned by `dladm_bridge_link_state`, using `bdcPortGetState`.

The new per-link bridging parameters are handled within the library as link properties. The running values of "stp", "stp_p2p", "stp_cost", "stp_edge", and "stp_priority" are accessed inside the library using an internal `dladm_bridge_get_port_cfg` function and `bdcPortGetConfig`.

The "stp_mcheck" property is special. In addition to the access described above, the property will be reset to 0 by the bridging daemon when the check has completed. The user will be unable to set this value to 1 when "force protocol" is 1 or less. This is done in order to fulfill the 802.1D administrative requirements.

The "forward" property is read with `dladm_bridge_get_forwarding`, which uses `bdcPortGetForwarding`. A separate interface is used because this property is not part of the librstp configuration structure.

The "default_tag" property is a new MAC property, `MAC_PROP_PVID`, and is accessed through the existing MAC property interfaces. (This is not part of the door interface, but is listed here for clarity.)

A new `dladm_get_linkprop_values` function retrieves these parameters individually for each link (based on `datalink_id_t`), and returns a simple integer.

2.1.2 Kernel Status Interface

The `dladm_bridge_get_fwdtable` function returns an array of structures representing a snapshot of the current kernel forwarding table for a bridge instance. This array is freed by `dladm_bridge_free_fwdtable`.

The forwarding table is read by opening the observability device node and issuing a `BRIOC_LISTFWD` ioctl repeatedly. This ioctl returns successive entries using the AVL iteration functions.

2.1.3 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_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 sends an SMF "refresh" to the daemon so that it will read the new values. 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 (via SMF "refresh") so that it can make corresponding changes.

When any of the new per-link bridging parameters are modified, the SMF "refresh" mechanism is used to notify the daemon.

Inside the daemon, when STP is disabled, the forwarding controls directly change the state of the port. Otherwise, they send a signal to the STP library to change port state.

When a port is removed from a bridge through the SMF interfaces, 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.

The daemon detects link, VLAN, and other configuration changes during refresh by walking the link database.

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 had not yet been ported to Solaris or OpenSolaris, but, after doing some simple porting and testing, it appears

to be functional. We have submitted it to UNH for interoperability testing, and it passed without exception. Conformance testing will also be performed.

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.

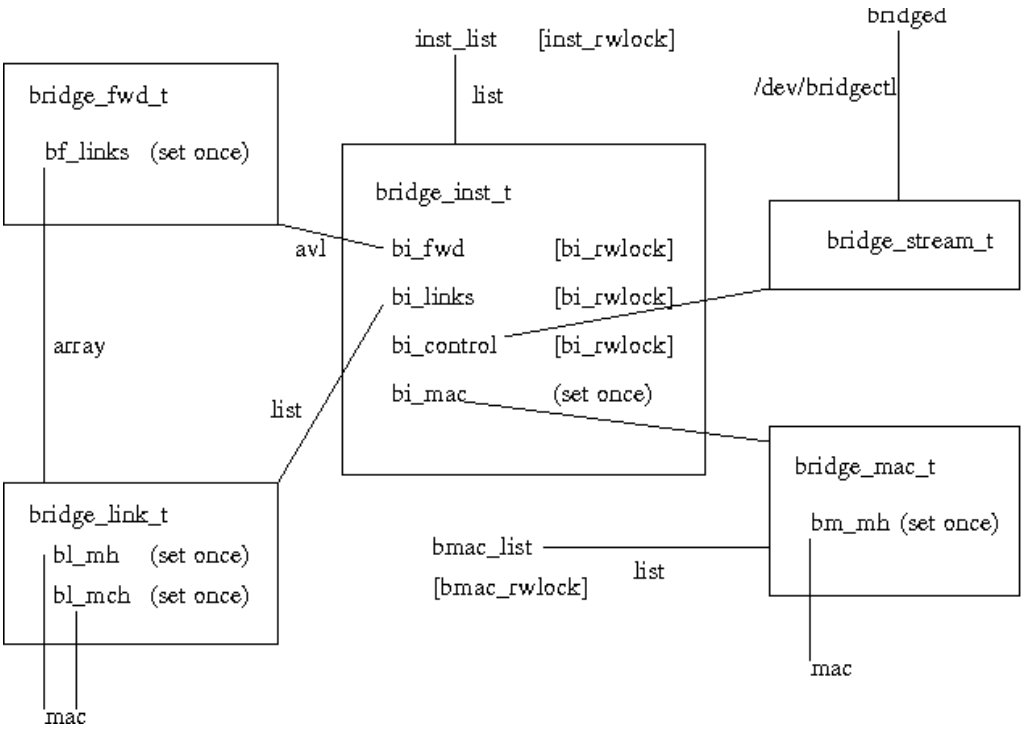
2.3 Kernel Bridging

2.3.1 Data Structures And Control

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 the global bridge instance list, 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.)

We support a list of links on the forwarding entries so that the degenerate case of multiple MAC entities with the same configured MAC address will function as expected; each getting a copy of matching datagrams.

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.

2.3.2 Crossbow Comparison

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

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

2. The Crossbow L2 entries are in a fixed-size 1024 entry hash table that uses a simple ad-hoc linked list (not using the common `sys/list.h`), while the bridge forwarding entries are in an AVL tree using the common kernel code.

3. The Crossbow entry identifies an output for delivery using a callback mechanism; the bridge forwarding mechanism identifies a list of output links.

4. Failing to find an entry in the Crossbow flow table 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. Addition and removal of Crossbow flows requires a blocking wait for threads to exit using locks that can (by design) be held only at process context level (`mac_flow_wait`), while bridge forwarding entries need to be added and removed quickly on learning new data and are designed to be removed from interrupt context.

For reference, in order to make Crossbow flow 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 flow table so that it stores entries per bridge rather than per link. (A likely design would be a cascaded approach: search link-local entries first, and, if that misses, then search the bridge-wide entries. Doing it in this order preserves performance for the "normal" case of input to a local VNIC.)

2. Enhance the Crossbow search mechanism so that it can handle the replication necessary when failing to locate an entry in the table, or at least make a callback to allow some other module to do this.

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 current Crossbow flow tables are analogous to the `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, and is tied to particular physical interfaces, while the other is used for routing, is ephemeral in nature, and is global to all interfaces. For this reason, we will not be reusing the Crossbow technology in the first release.

It would have been possible to base the L2 identification portion of Crossbow's flow table mechanism on the bridge forwarding mechanism, as bridge forwarding is at least done early enough in the process that it could identify a particular VNIC. A plausible way to implement this would be to allow the driver to hint at the VNIC receiver; hardware that can recognize multiple addresses and tag packets could use this to optimize performance with VNICs, and bridging could use it by setting the tag based on its forwarding lookup. However, given the likely and inherent performance issues with bridging (see the performance section later in this document), it seems quite unwise to use the bridging mechanism this way.

A future project may address this area. It would likely be useful to be able to apply Crossbow flow controls in bridge forwarding decisions, but it's currently unclear how to do that. One approach would be to modify the Crossbow design (as outlined above) to provide bridge-friendly features, and use that to replace this project's forwarding table. Another would be to create compatible (but separate) flow classification and control mechanisms within the bridge forwarding function provided here.

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 special bridge MAC observability node. This node will be created in the `/dev/net/` directory. Each bridge instance has a node (named by the bridge name plus a terminating "0" character to make it a legal DLPI driver name), 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 a later sub-

section of this document.)

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.

The administrator will not be able to create VNICs on top of the bridge observability nodes, due to the use of `mac_unicast_add` in the VNIC creation code. Because the observability nodes cannot be used for plumbing IP, the only plausible purpose for VNICs would be to place observability nodes into non-global zones. A future project could remove this restriction.

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

2.5 Link Up/Down Status

There are two significant parts to this. First, the bridge module needs to intercept the link up/down processing within the mac layer, so that the mac layer can return `LINK_STATE_UP` to the clients when at least one link on the bridge is up. The other part is to handle bridge link addition and removal, so that the system notifies the other link clients when an added link is the first one "up" (making the whole bridge "up") or when a removed link was the last one "up" (making the whole bridge down).

The interception logic consists of moving the real link state into a new `mi_lowlinkstate` variable, leaving the bridge composite state reported to clients in `mi_linkstate`, and establishing a `mac_bridge_ls_cb` callback for determining aggregate state. To allow the bridging module to broadcast link state changes (for example, when the last link goes down on a bridge) without fear of recursion, there is also a new `mac_link_redo` function. This will act like `mac_link_update`, but will not call the bridging module.

For link addition, simply marking the link initially as "down," adding it to the bridge, and then setting the real state causes the right things to happen. Link removal is a little harder. After marking the link as deleted, the bridge must check if the link being deleted was up, and if all of the remaining non-deleted links are down. If so, then the bridge broadcasts out a link-down event to all of those remaining links and their clients.

2.6 MTU Handling

The bridge MTU is taken from the first link that joins the bridge. If there's only one link on the bridge, then the bridge MTU follows that one link. If there multiple links on the bridge, then the bridge MTU remains fixed, and links that differ in MTU are not used.

The administrative experience is that links with the wrong MTU can be added to a bridge, but they don't become active until the MTU issue is corrected, and the "dladm show-bridge -l" command indicates that the link is disabled because the MTU is incorrect.

The bridge module does not disallow changes to a link's MTU, but it does react to them, and records them in `bridge_link_t.bl_maxsdu`. If there's only one link on the bridge, then changes to the link's MTU value will simply change the bridge's MTU (in `bridge_mac_t.bm_maxsdu`). If

there are multiple links, then any change away from the bridge's MTU by any link will cause the link to enter "bad SDU" state.

The "bad SDU" state is indicated by a `BLF_SDUFAIL` flag on `bl_flags`, so that the link is treated as "down" for bridging, and a `bridge_ctl_t` message is sent up the control stream to `bridged`. That message causes the daemon to disable the link with STP, just as though the link had gone down. The `mblk_t` for this message is preallocated in `bl_lfailmp`, so that the kernel can always report failures.

The bad SDU state is reported through `libdladm` to the dladm command for display with the "show-bridge -l" subcommand using a synthetic `UID_PORT_BADSDU` state value. This is a synthetic state; it uses flags in bridged, and doesn't come from `librstp` itself.

When the "mtu" link property is set on the bridge observability node, the kernel driver loops through the links on the bridge, and either sets or clears the "bad SDU" state based on the new MTU for the bridge. It's not intended that bridges will actually be administered this way, but it may be handy in transition.

There is also an interaction between this state and link up/down reporting. When a link is in "bad SDU" state, it behaves as though it's not really part of the bridge. This means that:

1. When figuring aggregate status for the bridge, the kernel need to treat "bad SDU" links as though they're down.

2. The link status on a "bad SDU" link should be the real, underlying link status, and not the aggregate bridge status. Only regular bridge links show the aggregate status.

3. When switching into and out of "bad SDU" state, the bridge module must update the mac layer about the link status, using either the real link status (when failed) or the aggregate bridge status (when normal).

The best way to administer MTU on links within a bridge is to disable the bridge first (using "svcadm disable -st bridge:*name*"), then set the MTU on all of the member links as desired, and then reenable the bridge ("svcadm enable -st bridge:*name*").

2.7 Debugging Support

In order to simplify debugging, we will create a "::dladm" dcmd for mdb. This will have a single subcommand to start, "show-bridge". It's expected that later projects will expand the list of subcommands to provide other common information.

The subcommand will support only the default (bridge instance list), "-l" (bridge link), and "-f" (bridge forwarding entry) options, and the display will include data structure addresses and other kernel debug related information, and will not include information contained only in the user-space daemon (such as STP state).

It's not intended that the "::dladm" subcommand will grow to have all of the features of the user space implementation. Instead, it will provide just the basics needed to display key kernel-resident information in a somewhat familiar (but not identical) format.

2.8 MAC Layer

There are several distinct changes to the MAC layer in support of bridging. These include a snoop-only MAC device, IEEE Port VLAN ID (PVID) support, and the bridging I/O hooks. The following sections describe these in detail.

2.8.1 Snoop-Only MAC Device

The snoop-only MAC device is implemented by a new `mac_no_active` function for drivers, that sets a new `MIS_NO_ACTIVE` on `mi_state_flags`. The `i_mac_unicast_add` function checks this flag for MAC clients, and returns `EBUSY` just as though the existing `MIS_EXCLUSIVE` were set.

2.8.2 PVID Support

The existing `mac_set_prop` function will test `mp_id` for `MAC_PROP_PVID`. If this property is being set, then it will call a new `mac_set_pvid` function. The existing `mac_get_prop` function will also check for the `mp_id` value, and will call a new `mac_get_pvid`.

The `mac_set_pvid` function loops through the list of MAC clients to determine if any are using the proposed Port VLAN ID. If any are, then it

returns EBUSY, as these packets are not transmitted with a VLAN tag. The mac_get_pvid just returns the current PVID value.

The i_mac_unicast_add function checks whether the client is asking to use the VLAN specified by mi_pvid (if any), and returns EBUSY if so. Clients must not attempt to use the PVID directly, and must instead use VLAN 0 for native encapsulation.

The new mi_pvid value is protected by the serializer.

The dls_link_header_info function will set a new mhi_ispvid flag when the PVID is seen on a packet. This allows any erroneously PVID-tagged packets to be detected by i_dls_link_rx.

2.8.3 Bridging I/O Hooks

The bridge module calls into the mac module (mac_bridge_vectors) to provide it with a set of call-back entry points for use with bridging. The call-backs are used only when already holding a reference to a bridge link, or when holding the mi_bridge_lock mutex and mi_bridge_link is non-NULL. These cases are sufficient to guarantee that the bridge module cannot unload during the execution of the call-back.

For the I/O hooks, we first simplify the mac implementation by removing the MAC_TX and MAC_RING_TX_DEFAULT macros. The only purpose these macros serve is to deal with the case where mi_default_tx_ring is NULL and call through the mi_tx function pointer. It's far simpler to call mac_ring_tx and let it call mi_tx in the appropriate case, and the difference in overhead is negligible; at most a few instructions. The benefit is centralizing the transmit path so that bridge interception is simple.

Thus, mac_ring_tx becomes a wrapper function. If a bridge is not present (mi_bridge_link is set to NULL), then it calls mac_bridge_tx directly. If a bridge is present, then it uses mac_bridge_ref_cb (with mi_bridge_lock held) to take a reference on the bridge link, and then the mac_bridge_tx_cb call-back to the bridge module is invoked. The bridge then calls mac_bridge_tx when it needs to transmit on a link. The new mac_bridge_tx function contains the original mac_ring_tx logic with a new check for a NULL mac_ring_handle_t so that it can call mi_tx (centralizing the old macros).

Similarly, mac_rx also becomes a wrapper function. If no bridge is present, then a tail call is made to mac_bridge_rx, which contains the original max_rx logic. If a bridge is present, then it takes a reference on

the bridge link, calls `mac_bridge_rx_cb`, and then drops the reference. The bridge then uses `mac_bridge_rx` to deliver any packets that are to be received locally.

By placing the hooks in `mac_ring_tx` and `mac_rx`, we get access to the packets at the lowest possible level in the MAC layer, directly atop any aggregation or NIC driver.

The `mac_bridge_set` function establishes a bridge on a link by setting `mi_bridge_link` (points to the handle for the bridge link) under the `mi_bridge_lock` mutex.

The `mac_bridge_clear` function clears the `mi_bridge_link` pointer back to `NULL`. Because a thread invoking one of the call-backs will hold the lock while taking a bridge link reference, and then hold the reference while calling back to the bridge module, the bridge instance and link data structures are guaranteed to be valid for all threads still executing after the clear function returns.

Both the `mac_bridge_set` and `mac_bridge_clear` functions also call `mac_capab_update`, which causes the clients to reset any cached capabilities and call again to update. The `i_mac_capab_get` function checks `mi_bridge_link` and, if the link is part of a bridge, returns `B_TRUE` for the zero-copy "incapability" and `B_FALSE` otherwise. This disables all MAC capabilities for links in a bridge.

The set/clear functions also call a new `mac_poll_state_change` function. This sets or clears a new `MIS_POLL_DISABLE` in `mi_state_flags`, and updates all of the clients with `mac_client_update_classifier`. The flag will cause the clients to disable polling mode when bridging is enabled. We expect that the forced promiscuous mode used in bridging will negate the performance advantages for polling mode, and that the complexity of adding hooks into the polling paths for bridging purposes will be prohibitive.

As a special case, the `mac_promisc_dispatch` function will skip over clients created by bridges, which have no registered receive function in `mpi_fn`. These clients are just used to hold the link open while doing I/O through the hooks defined above.

2.9 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.

When the user modifies bridge or link parameters, or adds or removes links from the bridge, libdladm will use the SMF "refresh" mechanism to cause the daemon to re-read its configuration. This will cause SIGHUP to be sent to the daemon, which will then call libscf to get new parameters, and walk over the links using `dladm_walk_datalink_id`.

2.9.1 Security

The daemon provides a private door interface that is used to extract live statistics from STP. Just read-only access is provided.

The kernel interfaces require `PRIV_SYS_NET_CONFIG` for access.

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

(The auditing area may have architectural weaknesses due to the existing lack of support 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.)

2.10 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.

IPoIB cannot be bridged with Ethernet, because it is a dissimilar MAC layer and does not work with the 802 mechanisms used in this project. If OpenSolaris eventually supports EoIB, then bridging with InfiniBand should become possible. (In fact, if it didn't work, that should be considered a bug.)

4 Unaddressed Issues And Future Work

We are providing no mechanism for users to manipulate the kernel forwarding entries. Although there's no equivalent of "route add" for bridging, users can view the forwarding entry list. It's possible that a future project may address forwarding entry add and delete.

We are not supporting the identification of flows forwarded through a bridge, because the bridge forwarding function currently takes place at a layer below the Crossbow flow tables. A future project may address this area.

Bridges cannot be created or controlled from within non-global zones. As with link aggregations, neither the necessary privileges nor the necessary objects are present inside a non-global zone. A future project that makes control of link aggregation possible within non-global zones should consider doing the same for bridging.

Administrators cannot create VNICs on top of the bridge observability nodes due to the internal operation of VNICs. A future project could remove this restriction.

This project will not deliver FCS-regeneration-free bridging.

A future project may provide additional protection mechanisms for the bridge learning function. This might include ESADI, 802.1X, and/or special "hold-down" algorithms for defending against forged MAC addresses from rogue nodes.

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.