Linux Bridge - Part 2

Posted **Jan 31, 2018** • Updated **Oct 26, 2024**

By **Hechao Li** **11 min** read

Overview

In previous post, I described the configuration of a linux bridge and showed an experiment in which wireshark was used to analyze the traffic. In this article, I will discuss what happens when a bridge is created and how a Linux bridge works.

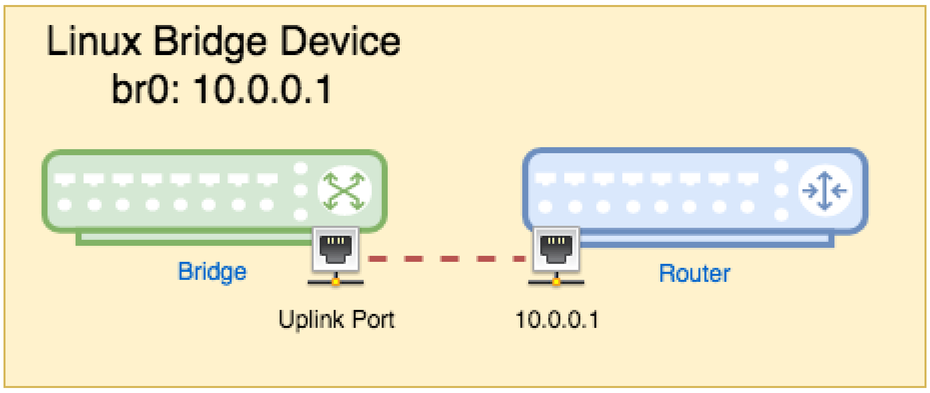
The source code related to bridge can be found here

Bridge Device

From *Understanding Linux Network Internals*:

> In Linux, a bridge is a virtual device. As such, it cannot receive or transmit anything unless you bind one or more real devices to it.

After reading last article, somebody asked me, since a bridge is a L2 device, why we see an IP address with it when we run `ifconfig br0`? The answer is, the Linux implementation combines a bridge and a router together. We know that a bridge has an uplink that can be connected to a router. Linux bridge makes this connection internal to the kernel. And this router is actually the real device it is bound to. We will discuss that part later. For understanding the bridge, we can just treat this IP address as the bridge's default gateway. And it is not needed if we want a private bridge.



Bridge Data Structure

The definition of the `net_bridge` struct can be found here.

Only some important fields are listed below:

```c
struct net_bridge
{
    spinlock_t          lock;
    struct list_head    port_list;
    struct net_device   *dev;

    spinlock_t          hash_lock;
    struct hlist_head   hash[BR_HASH_SIZE];
    bridge_id           bridge_id;
    ...
}
```

`port_list` is the list of ports a bridge has. Each bridge can have at most `BR_MAX_PORTS` (1024) ports. `dev` is a pointer to the `net_device` struct that represents the bridge device. `hash` is the forwarding table with `BR_HASH_SIZE` (256) entries.

Create a Bridge Device

A bridge can be created with command `brctl addbr br0`. It eventually calls `ioctl` with request `SIOCBRADDBR`:

```shell
# strace brctl addbr br0
execve("/sbin/brctl", ["brctl", "addbr", "br0"], [/* 17 vars */]) = 0
...
ioctl(3, SIOCBRADDBR, "br0")            = 0
...
```

If we search `SIOCBRADDBR` in the source code, we will see it is handled by function `br_ioctl_deviceless_stub`, along with 3 other requests - `SIOCGIFBR`, `SIOCSIFBR` and `SIOCBRDELBR`.

```c
int br_ioctl_deviceless_stub(struct net *net, unsigned int cmd, void __user
*uarg) {
    switch (cmd) {
    case SIOCGIFBR:
    case SIOCSIFBR:
        ...

    case SIOCBRADDBR:
    case SIOCBRDELBR:
        ...
    }
    ...
}
```

The function is registered in the initialization function `br_init`.

```c
int __init br_init(void) {
...
brioctl_set(br_ioctl_deviceless_stub);
...
}
```

`br_ioctl_deviceless_stub` calls `br_add_bridge`, which allocates a `net_device` that represents the bridge and then adds this device to the kernel interfaces using `register_netdev`.

```c
int br_add_bridge(struct net *net, const char *name)
{
    struct net_device *dev;
    int res;
    dev = alloc_netdev(sizeof(struct net_bridge), name, NET_NAME_UNKNOWN,
               br_dev_setup);
    ...
    res = register_netdev(dev);
    ...
    return res;
}
```

`alloc_netdev` is defined as a macro and it is the stub of `alloc_netdev_mqs`, which is in fact a generic method to allocate a `net_device` struct and is not specific to the bridge. The bridge is represented as private data in the `net_device`. (Private data is a piece of memory appended to the `net_device` struct.) The callback function that `alloc_netdev_mqs` takes is used to set up this private data part. In bridge case, the callback is `br_dev_setup`.

```c
struct net_device *alloc_netdev_mqs(..., void (*setup)(struct net_device *), ...) {
    struct net_device *dev;
    size_t alloc_size;
    ...
    alloc_size = sizeof(struct net_device);
    if (sizeof_priv) {
        /* ensure 32-byte alignment of private area */
        alloc_size = ALIGN(alloc_size, NETDEV_ALIGN);
        alloc_size += sizeof_priv;
    }
    /* ensure 32-byte alignment of whole construct */
    alloc_size += NETDEV_ALIGN - 1;
    p = kzalloc(alloc_size, GFP_KERNEL | __GFP_NOWARN | __GFP_REPEAT);
    ...
    dev = PTR_ALIGN(p, NETDEV_ALIGN);
    ...
    setup(dev);
    ...
    return dev
}
```

`br_dev_setup` sets up the private data area of the `net_device`. It casts this piece of memeory into `net_bridge` and initializes each field.

```c
void br_dev_setup(struct net_device *dev) {
    struct net_bridge *br = netdev_priv(dev);
    ...
    dev->priv_flags = IFF_EBRIDGE;
    br->dev = dev;
    spin_lock_init(&br->lock);
    INIT_LIST_HEAD(&br->port_list);
    spin_lock_init(&br->hash_lock);

    br->bridge_id.prio[0] = 0x80;
    br->bridge_id.prio[1] = 0x00;
    ...
}
```

To summarize, the call stack of creating a bridge is:

```
ioctl(3, SIOCBRADDBR, "br0")
 |- br_ioctl_deviceless_stub
     |- br_add_bridge
         |- alloc_netdev
             |- br_dev_setup
         |- register_netdev
```

After this step, we have a Linux bridge device. However, we do not have any interface bound to it, which means the bridge cannot transmit or receive anything yet.

Add an Interface

After the creation of a bridge, we can add interfaces (ports) to it. Though *Understanding Linux network internals* states that the bridge must be bound to a “real device”, I think this sentence is only partially true. It is true that a bridge device must be bound to a network interface. However, it does not necessarily need to be a “real device”, which means it does not need to be the interface of the real physical NIC. Even a tap interface can be bound to the bridge for it to function.

The command used to bind an interface is `brctl addif br0 tap0`. This is also called “enslave”, which means we are making `tap0` a slave of `br0`.

Shell

```shell
# strace brctl addif br0 tap0
execve("/sbin/brctl", ["brctl", "addif", "br0", "tap0"], [/* 17 vars */]) = 0
brk(NULL)                               = 0xf28000
...
ioctl(4, SIOCGIFINDEX, {ifr_name="tap0", }) = 0
close(4)                                = 0
ioctl(3, SIOCBRADDIF)                   = 0
```

It issues two `ioctl` requests - `SIOCGIFINDEX` and `SIOCBRADDIF`. The first one is used to search the index of interface `tap0` and the second one is used to add the interface `tap0` to the bridge. We only look into the second one.

`SIOCBRADDIF` is hanlded by `br_dev_ioctl`

C

```c
int br_dev_ioctl(struct net_device *dev, struct ifreq *rq, int cmd) {
    struct net_bridge *br = netdev_priv(dev);

    switch (cmd) {
    ...
    case SIOCBRADDIF:
    case SIOCBRDELIF:
        return add_del_if(br, rq->ifr_ifindex, cmd == SIOCBRADDIF);
    }
    ...
}
```

`add_del_if` is a stub for `br_add_if` and `br_del_if`. In addition case, `br_add_if` is called. Some important steps in `br_add_if` are listed below.

C

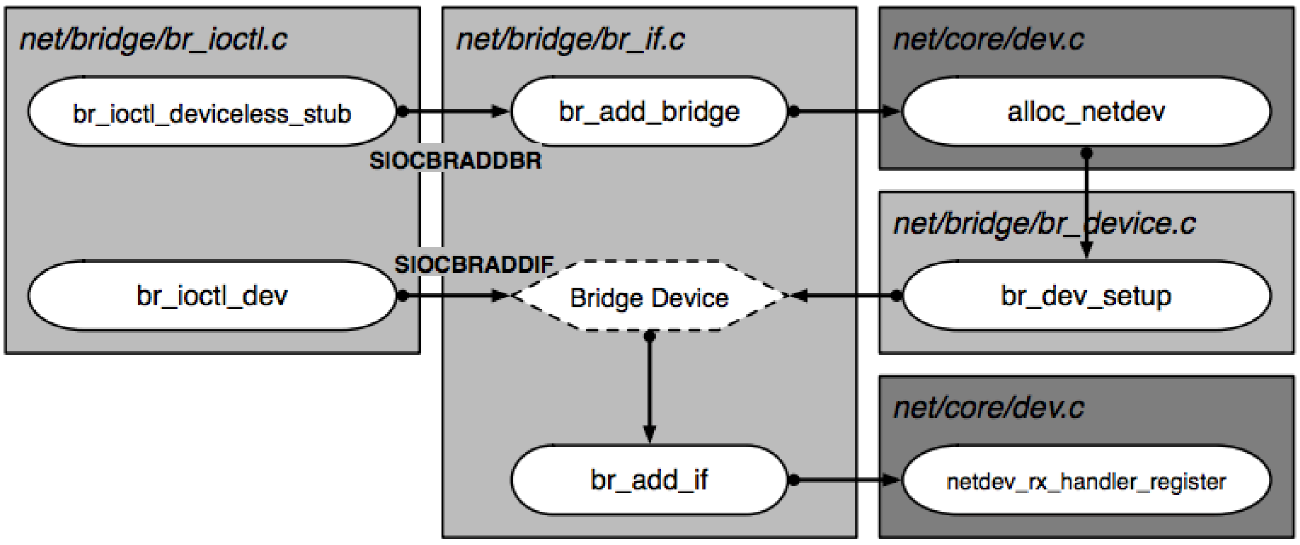
```c
int br_add_if(struct net_bridge *br, struct net_device *dev) {
    struct net_bridge_port *p;
    ... (Validation)
    p = new_nbp(br, dev);
    ...
    err = netdev_rx_handler_register(dev, br_handle_frame, p);
    ...
    err = netdev_master_upper_dev_link(dev, br->dev);
    ...
    list_add_rcu(&p->list, &br->port_list);
    nbp_update_port_count(br);
    ...
    if (br_fdb_insert(br, p, dev->dev_addr, 0))
        netdev_err(dev, "failed insert local address bridge forwarding table\n");
    ...
    return err;
}
```

Basically, `br_add_if` does the following things:

1. Perform a series of validations to make sure that this device can be enslaved to the bridge. Parts of the rules are: 1) non-ethernet like devices are not allowed; 2) a device that is already being enslaved is not allowed; 3) the device itself cannot be a bridge; 4) `IFF_DONT_BRIDGE` should not present in the device; etc.
2. Allocate and initialize a `net_bridge_port` struct. The port is later added to the bridge's `port_list`.
3. Register a receive handler `br_handle_frame` for the device. Frames sent to that device will be handled by this function. We will see what does this handler do later.
4. Enslave the device, which means making the bridge the master of this device.
5. Add this device's ethernet address to forwarding table as a local entry.

Note, in older version, the device is explicitly put into promiscuous mode in `br_add_if`. In 4.0 kernel, this is handled by `nbp_update_port_count` function.

The following figure from "Anatomy of a Linux bridge" shows the relationshiop of functioins we metioned above.



Forwarding Database

Forwarding database stores the MAC adress and port mapping. The implementation uses a hash table (an array actually) of size BR_HASH_SIZE(256) as the forwarding database. Each entry in the array stores the head of a singlely-linked list (a bucket) which stores the entries whose MAC address hash value falls into this bucket. See `struct hlist_head hash[BR_HASH_SIZE]` in the `net_bridge` sturct above. The hash value of the MAC address is calculted by `br_mac_hash`. I will skip the hash algorithm.

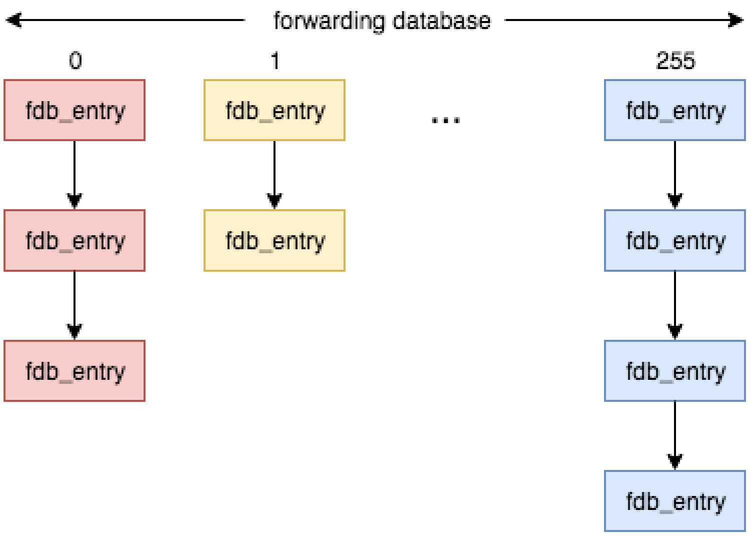


Table Entry Struct

`net_bridge_fdb_entry` is the element type of the aforementioned linked list.

```c
struct net_bridge_fdb_entry
{
    struct hlist_node       hlist;
    struct net_bridge_port  *dst;

    struct rcu_head         rcu;
    unsigned long           updated;
    unsigned long           used;
    mac_addr                addr;
    unsigned char           is_local:1,
                            is_static:1,
                            added_by_user:1,
                            added_by_external_learn:1;
    __u16                   vlan_id;
};
```

Lookup

Since the implementation is a hash table, lookup is same as any hash table lookup. The bridge looks up in the forwarding database when it wants to know which port to forward a frame with a particular MAC address. Thus the key of the hash table is the MAC address. First it gets the hash value of the MAC address by `br_mac_hash` and then gets the linked-list from that table entry. Next it iterates through the list and compare the MAC address with each element until it finds the right one.

Update Entries

The operations of adding, updating and deleting an entry are all typical hash table operations. I won't bother repeating how they work and only focus on when they happen.

- When an interface is added to the bridge (i.e. when `br_add_if` is called), `br_fdb_insert` is called to add the enslaved device's MAC address to the forwarding database.
- When a device on a local port changes its MAC address (For e.g. through `ifconfig eth0 ihw ether 11:22:33:44:55:66`), the entry is updated in the forwarding table.
- An entry is deleted when it expires. Each entry normally expires if it is not used for a while. By default it is 5 minutes but it's configurable. `br_fdb_cleanup` is called regularly to clean

up expired entries.

Frame Processing

Ingress data is first processed by `netif_receive_skb`, which is a generic device-independent function. It calls `rx_handler` of the device on which the data is received. Remember `br_add_if` registers a receive handler `br_handle_frame` for the enslaved device? Here this handler will be called to process data received on the device.

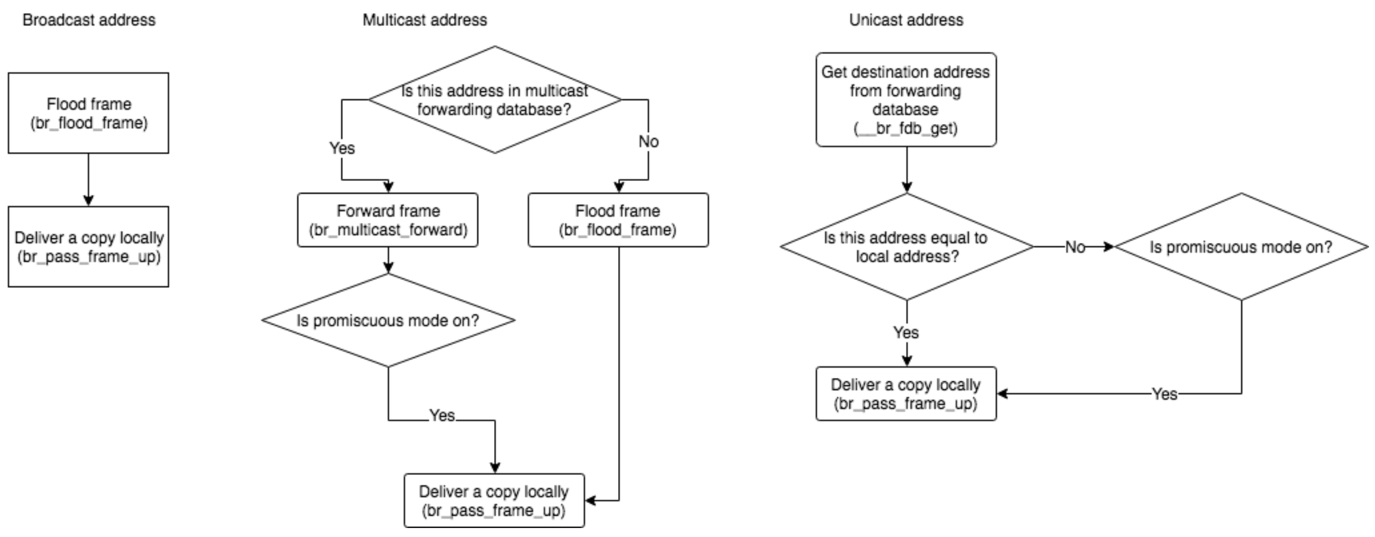
`br_handle_frame` does some sanity checking first, to ensure this frame is a valid Ethernet frame. Then it checks whether the destination is a reserved address, which means this is a control frame. Special processing is needed if the answer is yes. Otherwise it calls `br_handle_frame_finish` to handle this frame.

```c
rx_handler_result_t br_handle_frame(struct sk_buff **pskb) {
    const unsigned char *dest = eth_hdr(skb)->h_dest;
    ... (Validation code)
    if (unlikely(is_link_local_ether_addr(dest))) {
        /*
         * See IEEE 802.1D Table 7-10 Reserved addresses
         *
         * Assignment                 Value
         * Bridge Group Address       01-80-C2-00-00-00
         * (MAC Control) 802.3        01-80-C2-00-00-01
         * (Link Aggregation) 802.3 01-80-C2-00-00-02
         * 802.1X PAE address         01-80-C2-00-00-03
         *
         * 802.1AB LLDP         01-80-C2-00-00-0E
         *
         * Others reserved for future standardization
         */
        ... (Special processing for control frame)
    }

    ...
    NF_HOOK(NFPROTO_BRIDGE, NF_BR_PRE_ROUTING, skb, skb->dev, NULL,
            br_handle_frame_finish);
    ...
}
```

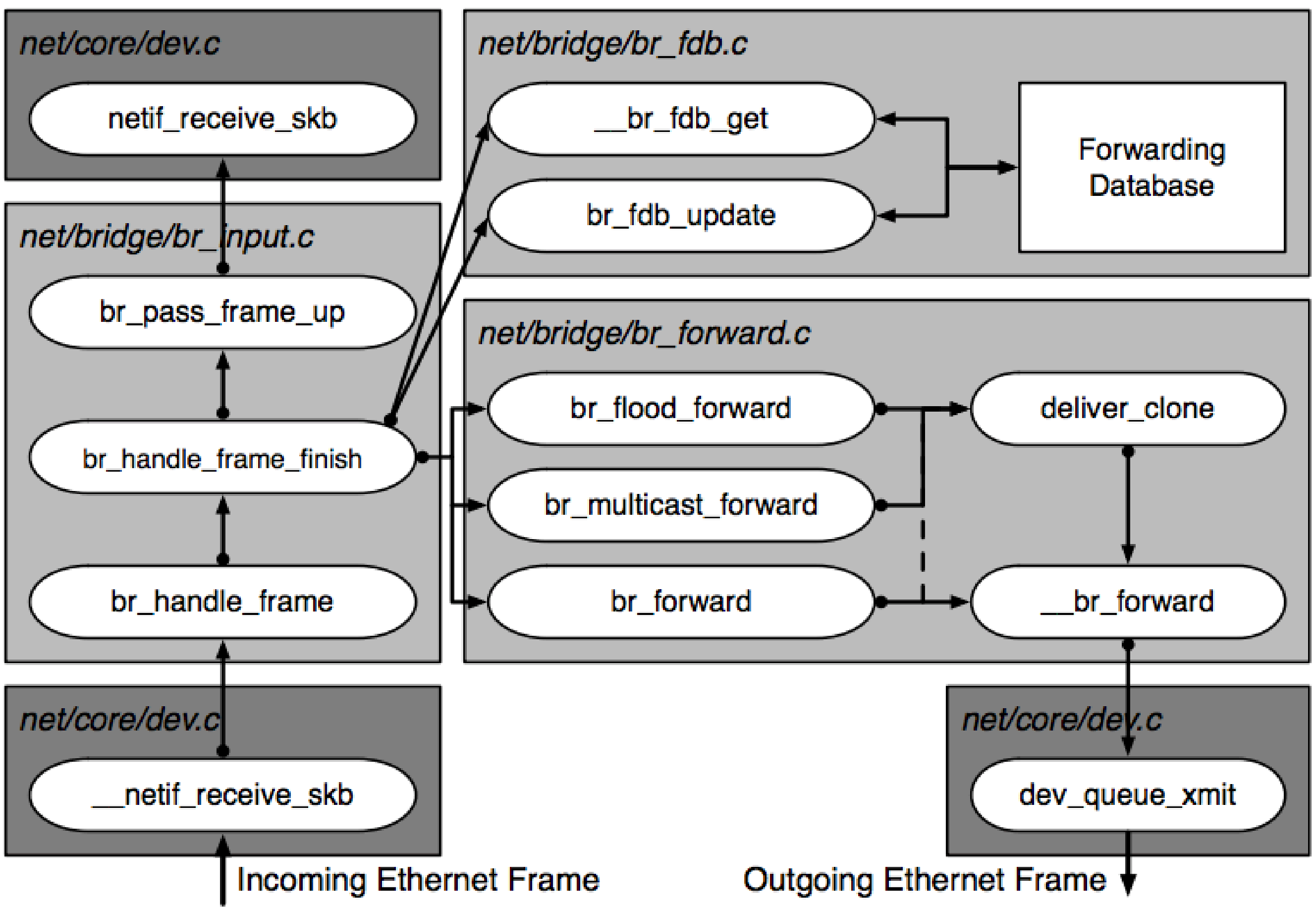
br_handle_frame_finish does the following steps:

1. Learn the source MAC address and update the forwarding database.
2. If the destination address is a multicast address, then `br_multicast_rcv` is called to do some processing.
3. The decision of how to forward this frame depends on whether the destination address is a broadcast, multicast or unicast address. The rules are as follows:


Broadcast address
Flood frame (br_flood_frame)
Deliver a copy locally (br_pass_frame_up)
Multicast address
Is this address in multicast forwarding database?
Yes
No
Forward frame (br_multicast_forward)
Flood frame (br_flood_frame)
Is promiscuous mode on?
Yes
Deliver a copy locally (br_pass_frame_up)
Unicast address
Get destination address from forwarding database (__br_fdb_get)
Is this address equal to local address?
No
Is promiscuous mode on?
Yes
Deliver a copy locally (br_pass_frame_up)
Yes

Note: if the interface is in promiscuous mode, then this frame will be delivered locally despite of the destination address. And promiscuous mode doesn't necessarily need to be on because anyway this bridge handler will forward it.

The following figure from "Anatomy of a Linux bridge" may also help understand the frame processing.



Conclusion

In this article, we discussed how Linux bridge is implemented and what happens when we configure the bridge as well as how a frame is processed by the bridge. Here is a lab that you can try. First add a bridge. Then start two VMs connected to this bridge. Check out the bridge's mac table by command `brctl showmacs br0`. And see if you can ping between two VMs. You can also use `tcpdump` or `Wireshark` to capture the traffic. Recall what happens when you do each step.

Reference

[1] Benvenuti, Christian. Understanding Linux network internals. " O'Reilly Media, Inc.", 2006.

[2] Anatomy of a Linux bridge

[3] Linux Bridge - how it works