Netfilter 之连接跟踪实现机制初步分析

2009-1
唐文
tangwen1123@163.com

1. 前言

Netfilter 中的连接跟踪模块作为地址转换等的基础，在对 Netfilter 的实现机制有所了解的基础上再深入理解连接跟踪的实现机制，对于充分应用 Netfilter 框架的功能和扩展其他的模块有着重大的作用。本文只是简要的分析连接跟踪的整体框架、其中的重要数据结构和重要函数，并粗略的描绘了数据包转发的连接跟踪流程。如果发现其中有问题欢迎及时与我联系。

2. 整体框架

连接跟踪机制是基于 Netfilter 架构实现的，其在 Netfilter 的不同钩子点中注册了相应的钩子函数，下面图 2-1 描绘了连接跟踪在 Netfilter 架构中注册的钩子函数。

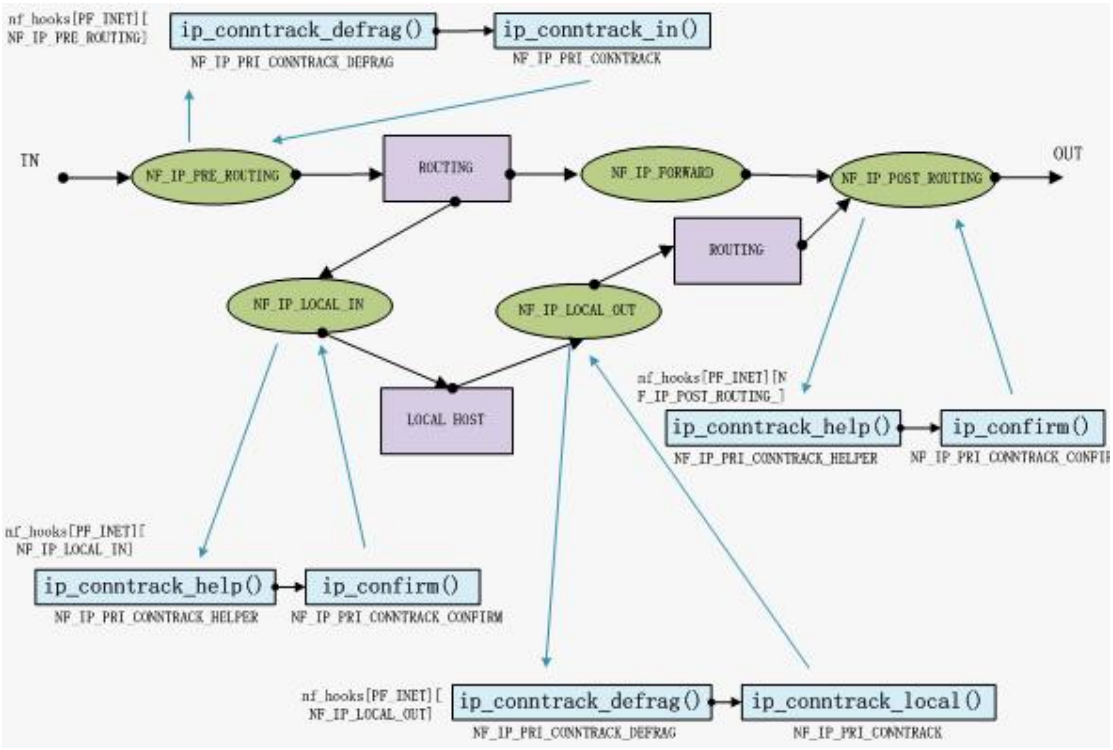

图 2-1 Netfilter 架构中的连接跟踪

3. 重要数据结构

3.1. 连接记录

在 Linux 内核中，连接记录由 ip_conntrack 结构表示，其结构如图 3-1 所示。在该结构中，包含一个 nf_conntrack 类型的结构，其记录了连接记录被公开应用的计数，也方便其他地方对连接跟踪的引用。每个连接记录都对应一个指向连接超时的函数指针，当较长时间内未使用该连接，将调用该指针所指向的函数。如果针对某种协议的连接跟踪需要扩展模块的辅助，则在连接记录中会有一指向 ip_conntrack_helper 结构体的指针。连接记录中的结构体 ip_conntrack_tuple_hash 实际记录了连接所跟踪的地址信息（源和目的地址）和协议的特定信息（端口）。所有连接记录的 ip_conntrack_tuple_hash 以散列形式保存在连接跟踪表中。

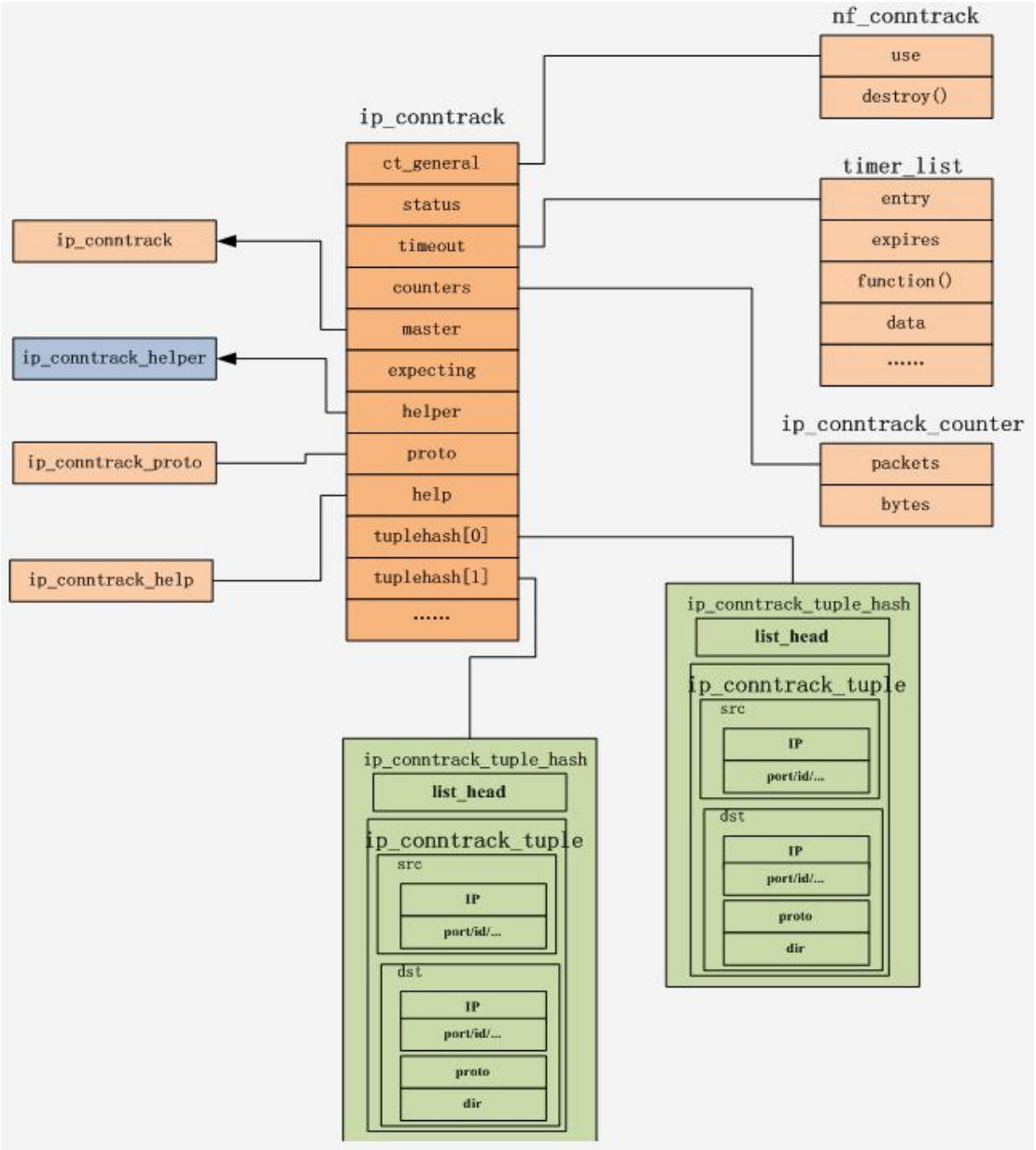

图 3-1 ip_conntrack 结构

3.2. 连接跟踪表

连接跟踪表是记录所有连接记录的散列表，其由全局变量 ip_conntrack_hash 所指向。连接跟踪表实际是一个以散列值排列的双向链表数组，链表中的元素即为连接记录所包含的 ip_conntrack_tuple_hash 结构，表的结构如下图 3-2 所示。

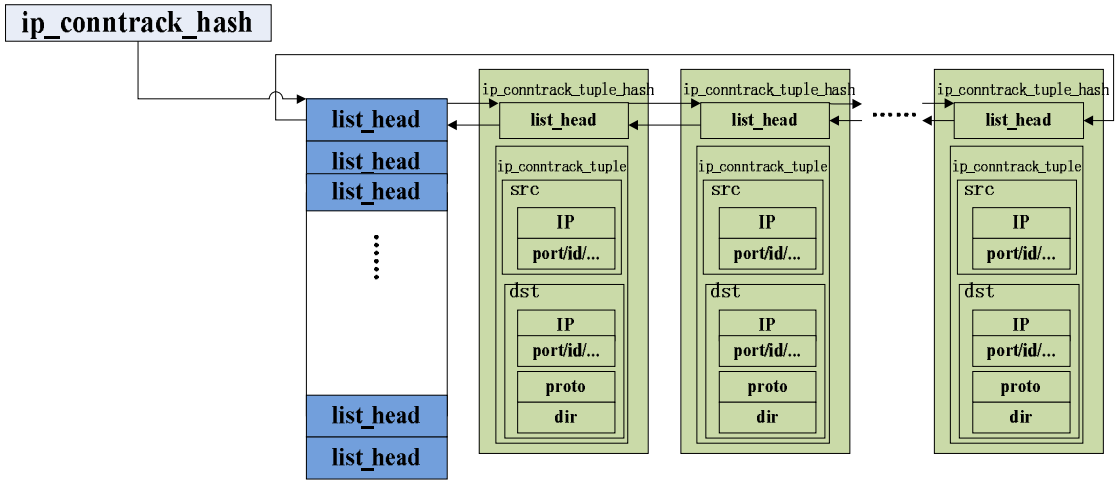

图 3-2 连接跟踪表

3.3. 连接跟踪辅助模块

在连接跟踪的实现机制中提供了 helper 辅助模块以扩展连接跟踪功能，一个辅助模块由一个结构体 ip_conntrack_helper 保存，该结构如下图 3-3 所示，所有注册的模块由全局变量 helpers 所指向的链表保存。函数 ip_conntrack_helper_register() 和 ip_conntrack_helper_unregister()用于在链表中添加和删除 ip_conntrack_helper 类型的结构。活动的 FTP 协议就使用了相应的 helper 模块来实现。

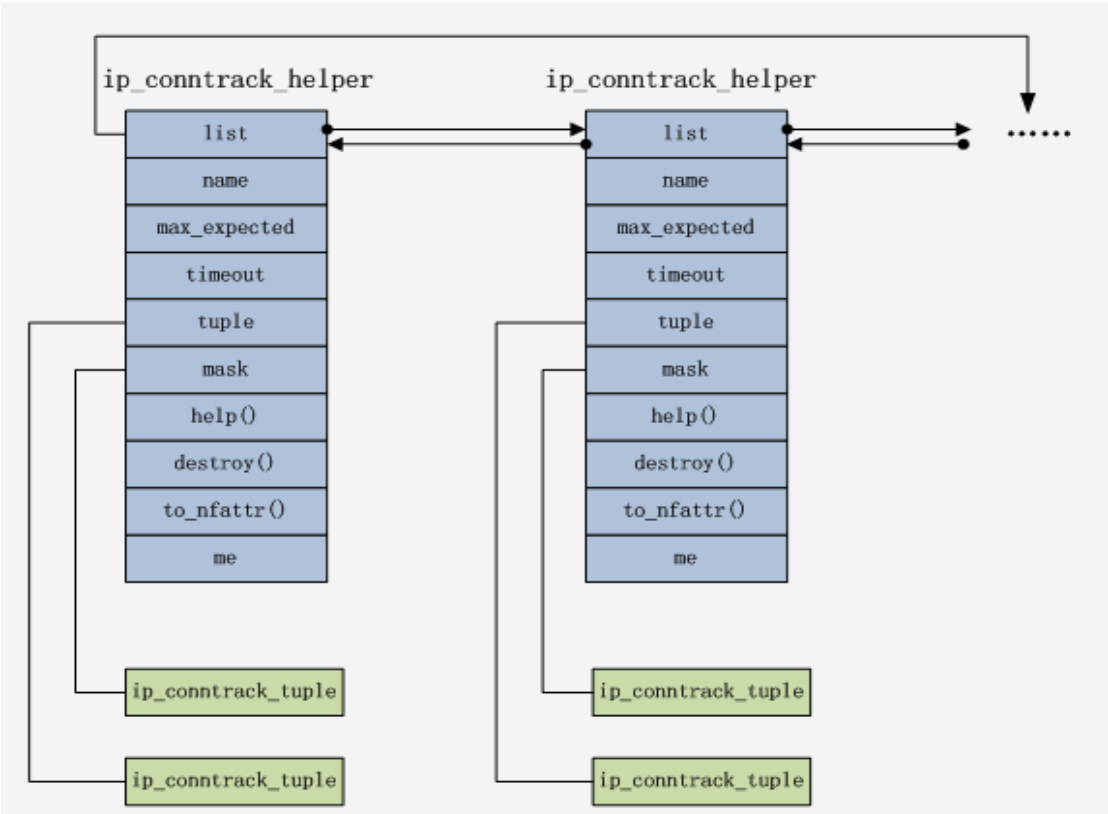

图 3-3 ip_conntrack_helper 结构

3.4. 期望连接

在连接跟踪机制中为了实现对活动协议的支持，还使用到了结构体 ip_conntrack_expect，其用于将预期连接分配给现有连接，有关于活动协议（如 FTP）的分析在此不做分析。ip_conntrack_expect 结构如下图 3-4 所示。所有的 ip_conntrack_expect 结构由全局变量 ip_conntrack_expect_list 指向的全局链表保存。

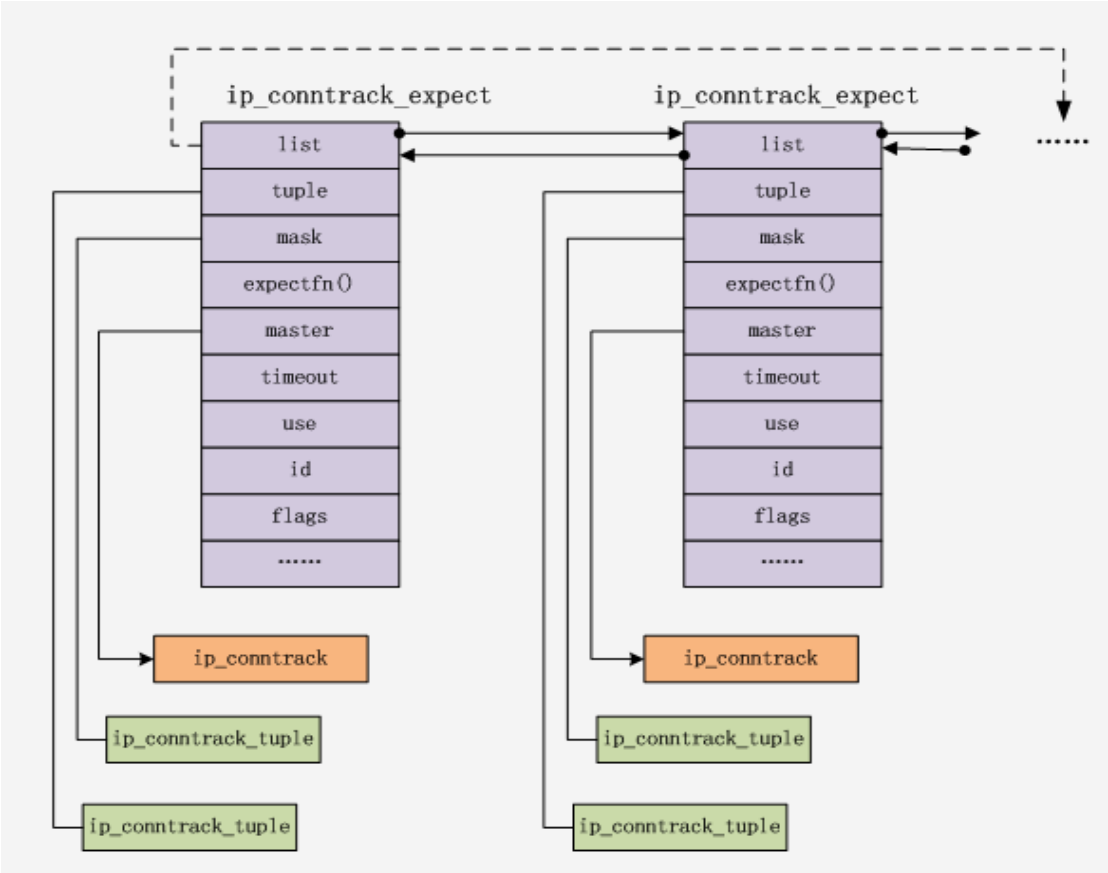

图 3-4 ip_conntrack_expect 结构

3.5. 传输协议

连接跟踪机制可以支持多种传输协议，不同的协议所采用的跟踪方式会有所不同。传输协议用结构 ip_conntrack_protocol 保存，所有的已注册的传输协议列表由全局变量 ip_ct_protos 所指向的一维数组保存，且按照协议号的顺序依次排列。函数 ip_conntrack_protocol_register()和 ip_conntrack_protocol_unregister()用于向协议列表中添加或删除一个协议。传输协议列表的结构如下图 3-5 所示。

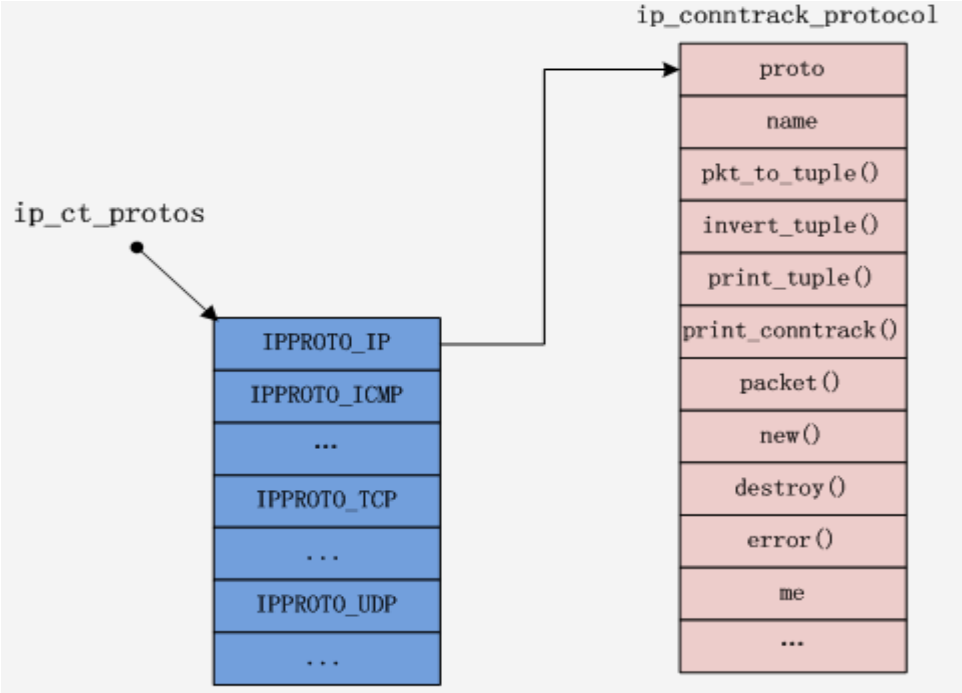

图 3-5 传输协议列表

4. 重要函数

4.1. ip_conntrack_defrag()

ip_conntrack_defrag()函数对分片的包进行重组，其调用 ip_ct_gather_frag()收集已经到达的分片包，然后再调用函数 ip_defrag()实现数据分片包的重组。ip_conntrack_defrag()被挂载在钩子点 NF_IP_PRE_ROUTING 和 NF_IP_LOCAL_OUT，即从外面进来的数据包或本地主机生成的数据包会首先调用该函数。该函数只操作数据包的内容，对连接跟踪记录没有影响，如果不需要进行重组操作则直接返回 NF_ACCEPT。函数的定义如下：

```
static unsigned int ip_conntrack_defrag(unsigned int hooknum,
                                        struct sk_buff **pskb,
                                        const struct net_device *in,
                                        const struct net_device *out,
                                        int (*okfn)(struct sk_buff *))
{
#if !defined(CONFIG_IP_NF_NAT) && !defined(CONFIG_IP_NF_NAT_MODULE)
        /* Previously seen (loopback)?    Ignore.    Do this before
            fragment check. */
        if ((*pskb)->nfct)
                return NF_ACCEPT;
#endif

        /* Gather fragments. */
        if ((*pskb)->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
                *pskb = ip_ct_gather_frags(*pskb,
                                           hooknum == NF_IP_PRE_ROUTING ?
                                           IP_DEFRAG_CONNTRACK_IN :
                                           IP_DEFRAG_CONNTRACK_OUT);
                if (!*pskb)
                        return NF_STOLEN;
        }
        return NF_ACCEPT;
}
```

4.2. ip_conntrack_in()

函数 ip_conntrack_in()被挂载在钩子点 NF_IP_PRE_ROUTING，同时该函数也被挂载在钩子点 NF_IP_LOCAL_OUT 的函数 ip_conntrack_local()调用，连接跟踪模块在这两个钩子点挂载的函数对数据包的处理区别仅在于对分片包的重组方式有所不同。

函数 ip_conntrack_in()首先调用__ip_conntrack_proto_find()，根据数据包的协议找到其应该使用的传输协议模块，接下来调用协议模块的 error()对数据包进行正确性检查，然后调用函数 resolve_normal_ct()在连接跟踪表中选择正确的连接跟踪记录，如果没有，则创建一个新纪录。接着调用协议模块的 packet()函数，如果返回失败，则 nf_conntrack_put()将释放连接记录。ip_conntrack_in()函数的源码如下，函数 resolve_normal_ct()实际操作了数据包和连接跟踪表的内容。

```
unsigned int ip_conntrack_in(unsigned int hooknum,
                             struct sk_buff **pskb,
                             const struct net_device *in,
                             const struct net_device *out,
                             int (*okfn)(struct sk_buff *))
{
        struct ip_conntrack *ct;
        enum ip_conntrack_info ctinfo;
        struct ip_conntrack_protocol *proto;
        int set_reply = 0;
        int ret;

        /* Previously seen (loopback or untracked)?    Ignore. */
        if ((*pskb)->nfct) {
                CONNTRACK_STAT_INC_ATOMIC(ignore);
                return NF_ACCEPT;
        }

        /* Never happen */
        if ((*pskb)->nh.iph->frag_off & htons(IP_OFFSET)) {
                if (net_ratelimit()) {
                printk(KERN_ERR "ip_conntrack_in: Frag of proto %u (hook=%u)\n",
                        (*pskb)->nh.iph->protocol, hooknum);
                }
                return NF_DROP;
        }

/* Doesn't cover locally-generated broadcast, so not worth it. */
#if 0
        /* Ignore broadcast: no `connection'. */
        if ((*pskb)->pkt_type == PACKET_BROADCAST) {
                printk("Broadcast packet!\n");
                return NF_ACCEPT;
        } else if (((*pskb)->nh.iph->daddr & htonl(0x000000FF))
                    == htonl(0x000000FF)) {
                printk("Should bcast: %u.%u.%u.%u->%u.%u.%u.%u (sk=%p, ptype=%u)\n",
                        NIPQUAD((*pskb)->nh.iph->saddr),
                        NIPQUAD((*pskb)->nh.iph->daddr),
                        (*pskb)->sk, (*pskb)->pkt_type);
        }
#endif

        /* rcu_read_lock()ed by nf_hook_slow */
        proto = __ip_conntrack_proto_find((*pskb)->nh.iph->protocol);
```

```
/* It may be an special packet, error, unclean...
         * inverse of the return code tells to the netfilter
         * core what to do with the packet. */
        if (proto->error != NULL
            && (ret = proto->error(*pskb, &ctinfo, hooknum)) <= 0) {
                CONNTRACK_STAT_INC_ATOMIC(error);
                CONNTRACK_STAT_INC_ATOMIC(invalid);
                return -ret;
        }

        if (!(ct = resolve_normal_ct(*pskb, proto,&set_reply,hooknum,&ctinfo))) {
                /* Not valid part of a connection */
                CONNTRACK_STAT_INC_ATOMIC(invalid);
                return NF_ACCEPT;
        }

        if (IS_ERR(ct)) {
                /* Too stressed to deal. */
                CONNTRACK_STAT_INC_ATOMIC(drop);
                return NF_DROP;
        }

        IP_NF_ASSERT((*pskb)->nfct);

        ret = proto->packet(ct, *pskb, ctinfo);
        if (ret < 0) {
                /* Invalid: inverse of the return code tells
                  * the netfilter core what to do*/
                nf_conntrack_put((*pskb)->nfct);
                (*pskb)->nfct = NULL;
                CONNTRACK_STAT_INC_ATOMIC(invalid);
                return -ret;
        }

        if (set_reply && !test_and_set_bit(IPS_SEEN_REPLY_BIT, &ct->status))
                ip_conntrack_event_cache(IPCT_STATUS, *pskb);

        return ret;
}
```

函数 resolve_normal_ct()搜索与传递来的 sk_buff 结构相匹配的连接记录，其首先调用函数 ip_ct_get_tuple()，利用包的协议模块的 pkt_to_tuple()函数创建一个 ip_conntrack_tuple 类

型的结构，接下来调用函数 ip_conntrack_find_get()在连接跟踪表中查找匹配的记录。如果没有找到匹配的项，将调用函数 init_conntrack()创建一个新的连接跟踪记录。最后确定数据包 sk_buff 结构的状态域的值，对其中的 nfct 和 nfctinfo 进行赋值。函数 resolve_normal_ct()的源码如下所示：

```
/* On success, returns conntrack ptr, sets skb->nfct and ctinfo */
static inline struct ip_conntrack *
resolve_normal_ct(struct sk_buff *skb,
                  struct ip_conntrack_protocol *proto,
                  int *set_reply,
                  unsigned int hooknum,
                  enum ip_conntrack_info *ctinfo)
{
        struct ip_conntrack_tuple tuple;
        struct ip_conntrack_tuple_hash *h;
        struct ip_conntrack *ct;

        IP_NF_ASSERT((skb->nh.iph->frag_off & htons(IP_OFFSET)) == 0);

        if (!ip_ct_get_tuple(skb->nh.iph, skb, skb->nh.iph->ihl*4,
                             &tuple,proto))
                return NULL;

        /* look for tuple match */
        h = ip_conntrack_find_get(&tuple, NULL);
        if (!h) {
                h = init_conntrack(&tuple, proto, skb);
                if (!h)
                        return NULL;
                if (IS_ERR(h))
                        return (void *)h;
        }
        ct = tuplehash_to_ctrack(h);

        /* It exists; we have (non-exclusive) reference. */
        if (DIRECTION(h) == IP_CT_DIR_REPLY) {
                *ctinfo = IP_CT_ESTABLISHED + IP_CT_IS_REPLY;
                /* Please set reply bit if this packet OK */
                *set_reply = 1;
        } else {
                /* Once we've had two way comms, always ESTABLISHED. */
                if (test_bit(IPS_SEEN_REPLY_BIT, &ct->status)) {
                        DEBUGP("ip_conntrack_in: normal packet for %p\n",
                               ct);
                        *ctinfo = IP_CT_ESTABLISHED;
                } else if (test_bit(IPS_EXPECTED_BIT, &ct->status)) {
                        DEBUGP("ip_conntrack_in: related packet for %p\n",
                               ct);
                        *ctinfo = IP_CT_RELATED;
```

```
} else {
                DEBUGP("ip_conntrack_in: new packet for %p\n",
                        ct);
                *ctinfo = IP_CT_NEW;
        }
        *set_reply = 0;
    }
    skb->nfct = &ct->ct_general;
    skb->nfctinfo = *ctinfo;
    return ct;
}
```

4.3. ip_conntrack_help()

函数 ip_conntrack_help() 被挂载在钩子点 NF_IP_LOCAL_IN 和 NF_IP_POST_ROUTING，其首先根据传来的 sk_buff 结构查找连接跟踪记录，如果该包所属连接有辅助模块 helper，且包符合一定的状态要求，则调用相应辅助模块的函数 help()处理数据包。

```
static unsigned int ip_conntrack_help(unsigned int hooknum,
                                      struct sk_buff **pskb,
                                      const struct net_device *in,
                                      const struct net_device *out,
                                      int (*okfn)(struct sk_buff *))
{
    struct ip_conntrack *ct;
    enum ip_conntrack_info ctinfo;

    /* This is where we call the helper: as the packet goes out. */
    ct = ip_conntrack_get(*pskb, &ctinfo);
    if (ct && ct->helper && ctinfo != IP_CT_RELATED + IP_CT_IS_REPLY) {
        unsigned int ret;
        ret = ct->helper->help(pskb, ct, ctinfo);
        if (ret != NF_ACCEPT)
            return ret;
    }
    return NF_ACCEPT;
}
```

4.4. ip_confirm()

函数 ip_confirm()被挂载在钩子点 NF_IP_LOCAL_IN 和 NF_IP_POST_ROUTING，其对数据包再次进行连接跟踪记录确认，并将新建的连接跟踪记录加到表中。考虑到包可能被过滤掉，之前新建的连接跟踪记录实际上并未真正加到连接跟踪表中，而在最后由函数 ip_confirm() 确认后真正添加，实际对传来的 sk_buff 进行确认的函数是__ip_conntrack_confirm()。在该函数中首先调用函数 ip_conntrack_get()查找相应的连接跟踪记录，如果数据包不是 IP_CT_DIR_ORIGINAL 方向的包，则直接 ACCEPT，否则接着调用 hash_conntrack()计算所找到的连接跟踪记录的 ip_conntrack_tuple 类型的 hash 值，且同时计算两个方向的值。然后根据这两个 hash 值分别查找连接跟踪记录的 hash 表，如果找到了，则返回 NF_DROP，如果未找到，则调用函数__ip_conntrack_hash_insert()将两个方向的连接跟踪记录加到 hash 表中。

```
static unsigned int ip_confirm(unsigned int hooknum,
                               struct sk_buff **pskb,
                               const struct net_device *in,
                               const struct net_device *out,
                               int (*okfn)(struct sk_buff *))
{
        /* We've seen it coming out the other side: confirm it */
        return ip_conntrack_confirm(pskb);
}
```

```
static inline int ip_conntrack_confirm(struct sk_buff **pskb)
{
        struct ip_conntrack *ct = (struct ip_conntrack *)(*pskb)->nfct;
        int ret = NF_ACCEPT;

        if (ct) {
                if (!is_confirmed(ct) && !is_dying(ct))
                        ret = __ip_conntrack_confirm(pskb);
                ip_ct_deliver_cached_events(ct);
        }
        return ret;
}
```

```
int
__ip_conntrack_confirm(struct sk_buff **pskb)
{
        unsigned int hash, repl_hash;
        struct ip_conntrack_tuple_hash *h;
        struct ip_conntrack *ct;
        enum ip_conntrack_info ctinfo;

        ct = ip_conntrack_get(*pskb, &ctinfo);

        /* ipt_REJECT uses ip_conntrack_attach to attach related
            ICMP/TCP RST packets in other direction.    Actual packet
            which created connection will be IP_CT_NEW or for an
            expected connection, IP_CT_RELATED. */
        if (CTINFO2DIR(ctinfo) != IP_CT_DIR_ORIGINAL)
                return NF_ACCEPT;

        hash = hash_conntrack(&ct->tuplehash[IP_CT_DIR_ORIGINAL].tuple);
        repl_hash = hash_conntrack(&ct->tuplehash[IP_CT_DIR_REPLY].tuple);

        /* We're not in hash table, and we refuse to set up related
            connections for unconfirmed conns.    But packet copies and
            REJECT will give spurious warnings here. */
        /* IP_NF_ASSERT(atomic_read(&ct->ct_general.use) == 1); */

        /* No external references means noone else could have
            confirmed us. */
        IP_NF_ASSERT(!is_confirmed(ct));
        DEBUGP("Confirming conntrack %p\n", ct);

        write_lock_bh(&ip_conntrack_lock);

        /* See if there's one in the list already, including reverse:
            NAT could have grabbed it without realizing, since we're
            not in the hash.    If there is, we lost race. */
        list_for_each_entry(h, &ip_conntrack_hash[hash], list)
                if (ip_ct_tuple_equal(&ct->tuplehash[IP_CT_DIR_ORIGINAL].tuple,
                                      &h->tuple))
                        goto out;
        list_for_each_entry(h, &ip_conntrack_hash[repl_hash], list)
                if (ip_ct_tuple_equal(&ct->tuplehash[IP_CT_DIR_REPLY].tuple,
                                      &h->tuple))
                        goto out;
```

```
/* Remove from unconfirmed list */
        list_del(&ct->tuplehash[IP_CT_DIR_ORIGINAL].list);

        __ip_conntrack_hash_insert(ct, hash, repl_hash);
        /* Timer relative to confirmation time, not original
           setting time, otherwise we'd get timer wrap in
           weird delay cases. */
        ct->timeout.expires += jiffies;
        add_timer(&ct->timeout);
        atomic_inc(&ct->ct_general.use);
        set_bit(IPS_CONFIRMED_BIT, &ct->status);
        CONNTRACK_STAT_INC(insert);
        write_unlock_bh(&ip_conntrack_lock);
        if (ct->helper)
                ip_conntrack_event_cache(IPCT_HELPER, *pskb);
#ifdef CONFIG_IP_NF_NAT_NEEDED
        if (test_bit(IPS_SRC_NAT_DONE_BIT, &ct->status) ||
            test_bit(IPS_DST_NAT_DONE_BIT, &ct->status))
                ip_conntrack_event_cache(IPCT_NATINFO, *pskb);
#endif
        ip_conntrack_event_cache(master_ct(ct) ?
                                 IPCT_RELATED : IPCT_NEW, *pskb);

        return NF_ACCEPT;

out:
        CONNTRACK_STAT_INC(insert_failed);
        write_unlock_bh(&ip_conntrack_lock);
        return NF_DROP;
}
```

4.5. ip_conntrack_local()

函数 ip_conntrack_local()被挂载在钩子点 NF_IP_LOCAL_OUT，该函数会调用 ip_conntrack_in()，函数源码如下：

```
static unsigned int ip_conntrack_local(unsigned int hooknum,
                                       struct sk_buff **pskb,
                                       const struct net_device *in,
                                       const struct net_device *out,
                                       int (*okfn)(struct sk_buff *))
{
        /* root is playing with raw sockets. */
        if ((*pskb)->len < sizeof(struct iphdr)
            || (*pskb)->nh.iph->ihl * 4 < sizeof(struct iphdr)) {
                if (net_ratelimit())
                        printk("ipt_hook: happy cracking.\n");
                return NF_ACCEPT;
        }
        return ip_conntrack_in(hooknum, pskb, in, out, okfn);
}
```

5. 数据包转发的连接跟踪流程

下面以数据包转发为例描述连接跟踪的流程，其中的函数及结构体为前几节所介绍的一部分，图中主要想体现数据包 sk_buff 在连接跟踪流程中的相应改变，连接跟踪记录与连接跟踪表的关系，何时查找和修改连接跟踪表，辅助模块以及传输协议如何在连接跟踪中使用等。所有的函数说明以及结构体在之前都有描述。发往本机以及本机发出的数据包的连接跟踪流程在此不再做分析。

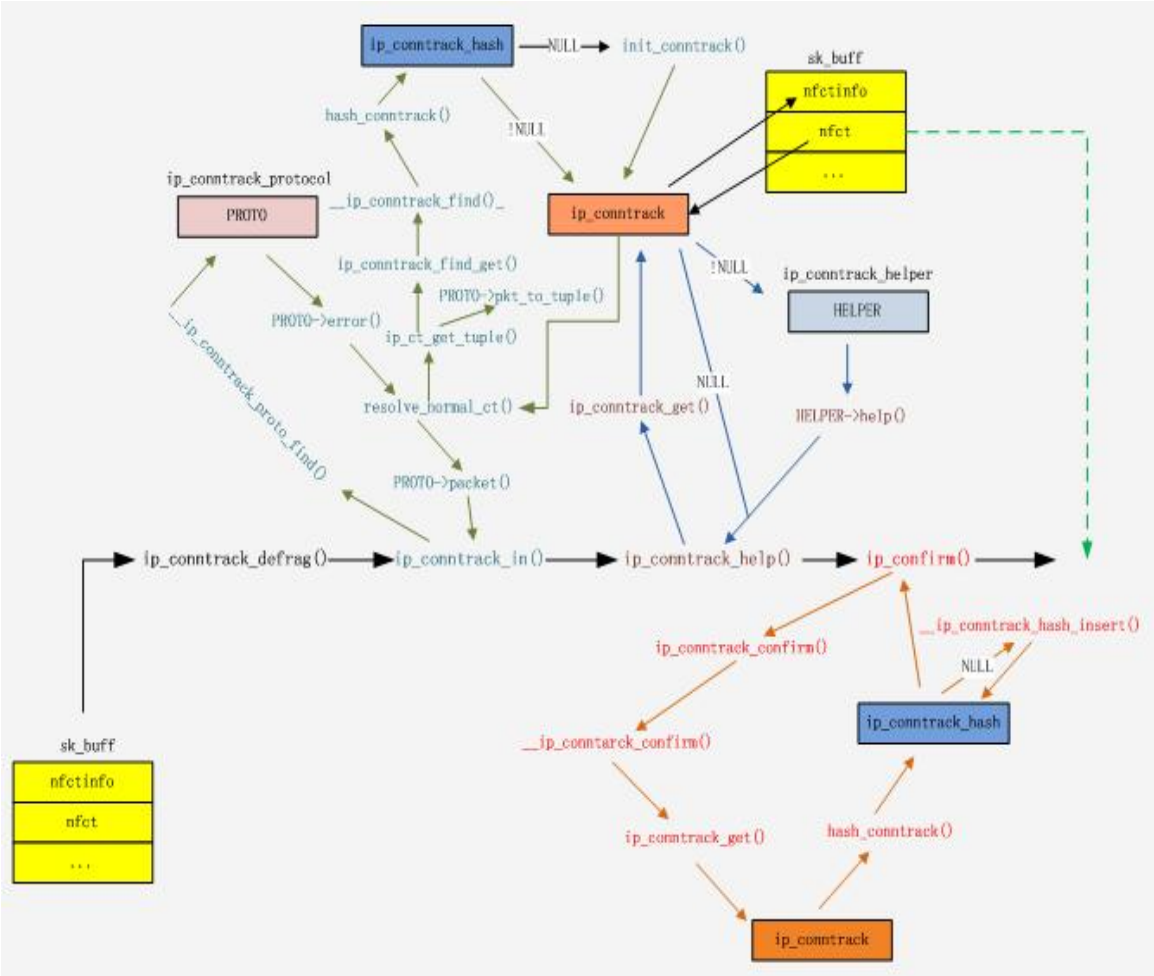

图 5-1 数据包转发的连接跟踪

6. 总结

以上只是简要分析了 Netfilter 架构中连接跟踪功能的实现机制，其中很多细节被忽略，如数据包的状态，连接跟踪记录的状态，具体传输协议的连接跟踪，主要目的是想要对整个实现框架有所认识。另外，对于较复杂的活动协议，期望连接与主连接之间的关联等并未做分析，希望在以后有时间再做分析。鉴于自身水平有限，且可参考资料较少，本文不能保证所描述的内容完全正确。