LVS源码分析(1)

由于要做一个类似LVS的包转发模块，研究了LVS的架构和代码，下面这个系列会做一个总结。首先推荐下这个blog http://yfydz.cublog.cn 里面对LVS, IPSec的讲解非常不错

几个重要的数据结构如下：

ip_vs_conn：一个连接由N元组构成，包括 caddr (客户端地址cip), vaddr (服务虚拟地址vip), daddr (目的realserver地址dip), cport (客户端连接端口), vport (服务虚拟端口), dport (目的realserver端口), protocol (协议)

ip_vs_service：代表一个虚拟服务。LVS中虚拟服务代表一个虚拟IP和端口，作为服务的入口，后面跟着一些realserver，在这些realserver之间做负载平衡。ip_vs_service中包括了protocol, addr, port。struct list_head destinations, __u32 num_dests则代表了后面realserver的链表和个数

ip_vs_dest：代表一个realserver。addr, port, weight分别代表了realserver的ip, port, 权重。struct dst_entry *dst_cache代表了从LVS到realserver的路由缓存项，在我看来这个应该只对NAT, tunnel模式有效。vport, vaddr, protocol代表了虚拟服务地址，端口和协议

ip_vs_scheduler：所有调度器的基类，对ip_vs_service进行调度，其最重要的方法是 struct ip_vs_dest* (*schedule)(struct ip_vs_service *svc, const struct sk_buff* skb)，从ip_vs_service下的ip_vs_dest数组中选取一个出来返回

static int __init ip_vs_init(void)用来初始化ipvs.ko，也就是LVS的核心模块：

ip_vs_control_init调用nf_register_sockopt注册struct nf_sockopt_ops结构，ip_vs_genl_register注册struct genl_ops ip_vs_genl_ops[]数组，这是通过netlink进行控制的命令结构。

ip_vs_protocol_init依次注册了ip_vs_protocol_tcp, ip_vs_protocol_udp, ip_vs_protocol_ah, ip_vs_protocol_esp四个协议

ip_vs_conn_init首先调用vmalloc分配一块大的内存(64k)区域用于存放连接的哈希表的key数组，也就是说有4096和list_head。

LVS最后调用nf_register_hooks，向netfilter注册自己的钩子结构。LVS一共有4个钩子(不算IPV6)，

```
static struct nf_hook_ops ip_vs_ops[] __read_mostly = {
    /* After packet filtering, forward packet through VS/DR, VS/TUN,
     * or VS/NAT(change destination), so that filtering rules can be
     * applied to IPVS. */
    {
        .hook       = ip_vs_in,
        .owner      = THIS_MODULE,
        .pf     = PF_INET,
        .hooknum        = NF_INET_LOCAL_IN,
        .priority       = 100,
    },
    /* After packet filtering, change source only for VS/NAT */
    {
        .hook       = ip_vs_out,
        .owner      = THIS_MODULE,
        .pf     = PF_INET,
        .hooknum        = NF_INET_FORWARD,
        .priority       = 100,
    },
    /* After packet filtering (but before ip_vs_out_icmp), catch icmp
     * destined for 0.0.0.0/0, which is for incoming IPVS connections */
    {
        .hook       = ip_vs_forward_icmp,
        .owner      = THIS_MODULE,
        .pf     = PF_INET,
        .hooknum        = NF_INET_FORWARD,
        .priority       = 99,
    },
```

```
    /* Before the netfilter connection tracking, exit from POST_ROUTING */
    {
        .hook       = ip_vs_post_routing,
        .owner      = THIS_MODULE,
        .pf     = PF_INET,
        .hooknum        = NF_INET_POST_ROUTING,
        .priority       = NF_IP_PRI_NAT_SRC-1,
    },

};
```

LVS无论是VS/DR, VS/TUN, VS/NAT哪种模式，由于vip配置在LVS上，因此访问vip的流量首先会走到NF_INET_LOCAL_IN，从而调用ip_vs_in

```
static unsigned int
ip_vs_in(unsigned int hooknum, struct sk_buff *skb,
     const struct net_device *in, const struct net_device *out,
     int (*okfn)(struct sk_buff *))
{

    ...

    // LVS ip_vs_in只处理发给本机的报文

    if (unlikely(skb->pkt_type != PACKET_HOST)) {
        IP_VS_DBG_BUF(12, "packet type=%d proto=%d daddr=%s
ignored\n",
                      skb->pkt_type,
                      iph.protocol,
                      IP_VS_DBG_ADDR(af, &iph.daddr));
        return NF_ACCEPT;
    }

    ...

    /*
     * Check if the packet belongs to an existing connection entry
```

```
     */

    // conn_in_get由协议本身实现，对于TCP而言，调用tcp_conn_in_get得到一个ip_vs_conn
    cp = pp->conn_in_get(af, skb, pp, &iph, iph.len, 0);

    if (unlikely(!cp)) {
        int v;


        /* For local client packets, it could be a response */
        cp = pp->conn_out_get(af, skb, pp, &iph, iph.len, 0);  // 查看是否是一个出去的连接
        if (cp)
            return handle_response(af, skb, pp, cp, iph.len); // 主要是执行snat


        if (!pp->conn_schedule(af, skb, pp, &v, &cp))  // 执行tcp_conn_schedule，TCP协议的调度就是为client找一个realserver，然后把这个conn保存下来，下次就直接基于这个ip_vs_conn转发了
            return v;
    }

    ...

    /* Check the server status */
    if (cp->dest && !(cp->dest->flags & IP_VS_DEST_F_AVAILABLE)) {
        /* the destination server is not available */
        if (sysctl_ip_vs_expire_nodest_conn) {
            /* try to expire the connection immediately */
            ip_vs_conn_expire_now(cp);
        }
        /* don't restart its timer, and silently
           drop the packet. */
        __ip_vs_conn_put(cp);  // 如果后面的realserver失效，那么drop这个ip_vs_conn
        return NF_DROP;
    }
```

```
    ip_vs_in_stats(cp, skb);
    restart = ip_vs_set_state(cp, IP_VS_DIR_INPUT, skb, pp);  // 调用
tcp_state_transition，改变连接的自动机状态
    if (cp->packet_xmit)
        ret = cp->packet_xmit(skb, cp, pp);  // 根据模式不同，调用不同的发
送方法 e.g. NAT调用ip_vs_nat_xmit, DR调用ip_vs_dr_xmit
        /* do not touch skb anymore */
    else {
        IP_VS_DBG_RL("warning: packet_xmit is null");
        ret = NF_ACCEPT;
    }

    ...

}
```

LVS的VS/DR, VS/TUN都是单臂模式，只有VS/NAT是双臂模式。在VS/NAT模式下，LVS会作为realserver回包的next hop，因此在NF_IP_FORWARD上注册ip_vs_out，用来处理NAT模式下的回包

```
static unsigned int
ip_vs_out(unsigned int hooknum, struct sk_buff *skb,
      const struct net_device *in, const struct net_device *out,
      int (*okfn)(struct sk_buff *))
{
    struct ip_vs_iphdr iph;
    struct ip_vs_protocol *pp;
    struct ip_vs_conn *cp;
    int af;

    ....

    ip_vs_fill_iphdr(af, skb_network_header(skb), &iph);  // 填充ip_vs_iphdr
的IP头

    if (unlikely(iph.protocol == IPPROTO_ICMP)) {  // 这部分代码用来处理
icmp报文，主要逻辑在ip_vs_out_icmp上，该函数用来处理outgoing方向
```

的icmp
```
        int related, verdict = ip_vs_out_icmp(skb, &related);

        if (related)
            return verdict;
        ip_vs_fill_iphdr(af, skb_network_header(skb), &iph);
    }

    ....

    if (unlikely(ip_hdr(skb)->frag_off & htons(IP_MF|IP_OFFSET) && !pp->dont_defrag)) {  // 如果是IP分片的包，那么调用ip_vs_gather_frags先尝试整合成一个完整包，具体请参考内核IP层的frag/defrag的相关代码
        if (ip_vs_gather_frags(skb, IP_DEFRAG_VS_OUT))
            return NF_STOLEN;

        ip_vs_fill_iphdr(af, skb_network_header(skb), &iph);
    }
```

```
    /*
     * Check if the packet belongs to an existing entry
     */
    cp = pp->conn_out_get(af, skb, pp, &iph, iph.len, 0);  // 查找是否有已有连接

    if (unlikely(!cp)) {
        if (sysctl_ip_vs_nat_icmp_send &&
            (pp->protocol == IPPROTO_TCP ||
             pp->protocol == IPPROTO_UDP)) {
            __be16 _ports[2], *pptr;

            pptr = skb_header_pointer(skb, iph.len,
                                      sizeof(_ports), _ports);
            if (pptr == NULL)
                return NF_ACCEPT;   /* Not for me */
            if (ip_vs_lookup_real_service(af, iph.protocol,
```

```
                    &iph.saddr,
                    pptr[0])) {   // 查看这个realserver是否在LVS的hash表
中，如果是真实的realserver，返回一个ICMP不可达

            /*
             * Notify the real server: there is no
             * existing entry if it is not RST
             * packet or not TCP packet.
             */
            if (iph.protocol != IPPROTO_TCP
                || !is_tcp_reset(skb, iph.len)) {
                    icmp_send(skb,
                        ICMP_DEST_UNREACH,
                        ICMP_PORT_UNREACH, 0);
                return NF_DROP;
            }
        }
    }
        IP_VS_DBG_PKT(12, pp, skb, 0,
                "packet continues traversal as normal");
        return NF_ACCEPT;
    }

    return handle_response(af, skb, pp, cp, iph.len);  // handle_response真
正去做SNAT
}

static unsigned int
handle_response(int af, struct sk_buff *skb, struct ip_vs_protocol *pp,
        struct ip_vs_conn *cp, int ihl)
{

    if (!skb_make_writable(skb, ihl))  /* 如果要修改skb的话，当前内核版本
需要先判断skb_make_writable */
        goto drop;
```

```
    /* mangle the packet */
    if (pp->snat_handler && !pp->snat_handler(skb, pp, cp))  /* 对TCP而言，这里是调用tcp_snat_handler, 主要功能是修改了tcp头之后再做下checksum */
        goto drop;

    ip_hdr(skb)->saddr = cp->vaddr.ip;  /* SNAT, 把包的源IP替换为virtual IP */
    ip_send_check(ip_hdr(skb));  /* 对IP头做checksum */

    /* For policy routing, packets originating from this

     * machine itself may be routed differently to packets
     * passing through.  We want this packet to be routed as
     * if it came from this machine itself.  So re-compute
     * the routing information.
     */

        if (ip_route_me_harder(skb, RTN_LOCAL) != 0)  /* 由于源IP变成了本地IP，而不是之前的转发包，需要重新计算路由 */
            goto drop;

    ip_vs_out_stats(cp, skb);
    ip_vs_set_state(cp, IP_VS_DIR_OUTPUT, skb, pp); /* 对TCP而言， 调用tcp_state_transition */
    ip_vs_conn_put(cp);

    skb->ipvs_property = 1;  /* 标记这个skb已经被LVS处理过 */

    LeaveFunction(11);
    return NF_ACCEPT;

drop:
    ip_vs_conn_put(cp);
    kfree_skb(skb);
    return NF_STOLEN;
```

```
}
```

LVS在NF_INET_POST_ROUTING chain上还注册了一个优先级为NF_IP_PRI_NAT_SRC - 1的hook函数ip_vs_post_routing。该函数在iptables SNAT之前执行，检查LVS是否处理过该skb，如果处理过则跳过下面的netfilter hook点

```
/*
 *      It is hooked before NF_IP_PRI_NAT_SRC at the
NF_INET_POST_ROUTING
 *      chain, and is used for VS/NAT.
 *      It detects packets for VS/NAT connections and sends the packets
 *      immediately. This can avoid that iptable_nat mangles the packets
 *      for VS/NAT.
 */
static unsigned int ip_vs_post_routing(unsigned int hooknum,
                       struct sk_buff *skb,
                       const struct net_device *in,
                       const struct net_device *out,
                       int (*okfn)(struct sk_buff *))
{
    if (!skb->ipvs_property)
        return NF_ACCEPT;
    /* The packet was sent from IPVS, exit this chain */
    return NF_STOP;
}
```

netfilter框架下，NF_HOOK宏会调用到nf_hook_slow，进而调用nf_iterate，即对于特定PF下的特定HOOKNUM，按优先级遍历上面注册的所有hook函数，只有当所有函数都返回NF_ACCEPT，或者有任意函数返回NF_STOP，整个nf_iterate才会返回NF_ACCEPT。与NF_ACCEPT不同的是，NF_STOP的语义会忽略该挂载点下其他优先级的函数。