连线跟踪学习总结

刘金成

一. 连线跟踪概述

连线跟踪条目(conntrack entry)用于记录linux在发送、转发、接收包时的状态信息。

在现实组网中，Router为PC分配一个私有地址192.168.1.5，PC使用这个私有地址向外发包，但是在Internet上源地址为192.168.1.5这样包是无法使用的，因此需要router在转发包的时候，首先将包的源地址修改为WAN地址136.1.1.24。当router收到internet的回应包时，它需要将其目的地址修改成私有地址192.168.1.5。这就是SNAT。但是这其中隐含一个问题，router收到回应包时，怎么知道应该将其目的地址修改为192.168.1.5而不是其它呢？方法是Router在转发包的时候，为它创建一个连线跟踪条目，并在router内存中保存一段时间。当router收到一个数据包的时候，就会将它和连线跟踪条目进行比对，如果比对成功，则会从连线跟踪条目中取出目的地址信息修改数据包头部。

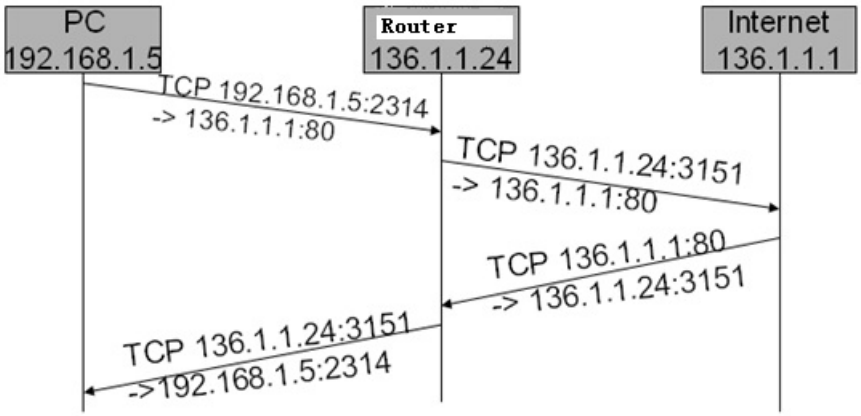


图1 NAT交互过程

Linux在数据包进入IP层协议栈的时候，即创建连线跟踪条目，并在协议栈内部持续记录各项信息。Linux可以在协议栈转发过程中利用这些信息修改包的内容，记录包 (log)，甚至丢弃一个包。比如可以配置一些规则，在数据包进入协议栈的时候丢弃一些包，在协议栈转发时记录包的信息，或者在离开协议栈时修改包的源地址，如前所述。

虽然一个数据包在IP层需要经过很多模块，但根据路由选择结果，及是否本机相关，共包括五个地方，如下图所示。

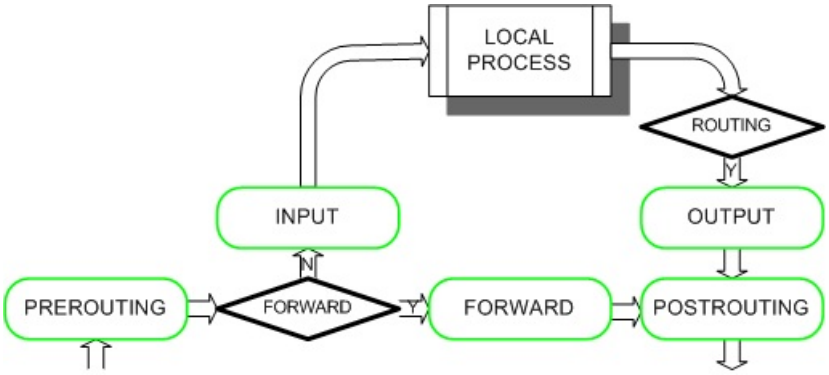


图2 Netfilter在L3的HOOK点

一个数据包可以用五元组<源IP、目的IP、传输层协议、源端口、目的端口>来唯一性地标记。协议栈在收到数据包时，使用这个五元组查找对应的连线跟踪条目。在Router看来，一条流有orig和reply两个方向，因此连线跟踪条目中需要记录两个方向的五元组信息。

除了手动配置一些iptables规则应用连线跟踪数据处理包外，linux协议栈还会使用连线跟踪管理一些有状态传输层协议（如TCP）的状态信息。比如在TCP三次握手没有完成时，收到该连接的一个数据包，则应该被视为异常，将包丢弃。

在传输层，可以使用连线跟踪管理状态信息。在应用层，连线跟踪同样有用。比如FTP协议，LAN侧的FTP客户端向WAN侧发送控制报文，同时在报文的应用层中约定双方使用的数据端口。服务器端在接收到这个控制报文后，读取数据端口，然后使用该数据端口向客户端发起TCP连接。注意服务器发起TCP连接时使用的五元组在Router中并没有记录，Router也就不知道怎么修改它的目的地址（SNAT的后半部），从而将包丢弃。为解决这个问题，需要Router在转发FTP控制报文的时候，穿透到该报文的应用层协议中，读取双方约定的数据端口，并使用该数据端口创建一个helper的连线跟踪条目。这就是ALG。

从上面的描述可以看到，linux中连线跟踪应用类型多（NAT、Mangle、Filter、ALG、Session管理），涉及范围广（LOCAL_IN、FORWARD、LOCAL_OUT、PRE_ROUTING、POST_ROUTING），涉及协议多（网络层、传输层、应用层），因此有必要使用一个专门的模块来管理。Linux中，这个模块是Netfilter。

二. Netfilter框架

1. NF_HOOK宏

在内核协议栈中，NF_HOOK是netfilter的入口。数据包在进入到IP层后，ip_rcv函数对包进行各项校验，校验完成调用ip_rcv_finish查询包的路由，根据路由查询结果决定本地转发报文，或者将报文交给本机更上层的协议。如图2所示。现通过NF_HOOK在包校验结束后，查询路由之前增加名为PRE_ROUTING的hook点，从这个入口进入连线跟踪模块处理包。

NF_HOOK(PF_INET, NF_INET_PRE_ROUTING, skb, dev, NULL, ip_rcv_finish);

PF_INET表示协议号，可以为INET、INET6等。

NF_INET_PRE_ROUTING表示hook点。

Skb表示待处理的数据包。

Dev表示收到这个包的netdevice。

NULL表示这个包将要从哪个netdevice出去，这里为空。

Ip_rcv_finish表示从netfilter模块出来后将要调用的函数。

```
#define NF_HOOK(pf, hook, skb, indev, outdev, okfn) \
    NF_HOOK_THRESH(pf, hook, skb, indev, outdev, okfn, INT_MIN)
```

增加一个参数INT_MIN，表示进入netfilter的最小优先级。通常为INT_MIN，表示该hook所有条目都会被执行。

```
#define NF_HOOK_THRESH(pf, hook, skb, indev, outdev, okfn, thresh)      \
({int __ret;                                                \
if ((__ret=nf_hook_thresh(pf, hook, (skb), indev, outdev, okfn, thresh, 1)) == 1)\
    __ret = (okfn)(skb);                                    \
__ret;})
```

进入Netfilter模块。注意只有返回值为1时才会执行okfn中的动作。意味着NF_HOOK可以将包释放，中断正常收包流程。只要模块返回0即可。

nf_hook_thresh表示的最后一个参数“1”，表示可以根据条件决定是否进入netfilter模块。例如可以对nf_hook_thresh作一个包装，NF_HOOK_THRESH_COND，再满足一定条件时才进入Netfilter。可以在收包运行期间判断，灵活性较强。

```
static inline int nf_hook_thresh(u_int8_t pf, unsigned int hook,
                 struct sk_buff *skb,
                 struct net_device *indev,
                 struct net_device *outdev,
                 int (*okfn)(struct sk_buff *), int thresh,
                 int cond)
{
    if (!cond)
        return 1;
#ifndef CONFIG_NETFILTER_DEBUG
    if (list_empty(&nf_hooks[pf][hook]))
        return 1;
#endif
```

根据list_empty(&nf_hooks[pf][hook])可以判断，Netfilter用一个二维表组织所有的nf条目。第一个维是协议，比如IPV4、IPV6、ARP、BRIDGE。第二维为HOOK，IP层的HOOK布局如图2所示。二维表中每一项都是一个list，顾名思义，从各个hook位置进入到Netfilter后，都会遍历这个list，检查skb。

```
    return nf_hook_slow(pf, hook, skb, indev, outdev, okfn, thresh);
}
```

如前所述，在nf_hook_slow中会遍历每个hook对应的list，逐个检查skb。

```
int nf_hook_slow(u_int8_t pf, unsigned int hook, struct sk_buff *skb,
         struct net_device *indev,
         struct net_device *outdev,
         int (*okfn)(struct sk_buff *),
         int hook_thresh)
{…
elem = &nf_hooks[pf][hook];
    verdict = nf_iterate(&nf_hooks[pf][hook], skb, hook, indev,
                 outdev, &elem, okfn, hook_thresh);
    if (verdict == NF_ACCEPT || verdict == NF_STOP) {
        ret = 1;
    } else if (verdict == NF_DROP) {
        kfree_skb(skb);
        ret = -EPERM;
…
}
```

注意，nf_iterate遍历的结果有可能为ACCEPT和DROP

以上为NF_HOOK的执行过程。每一个hook入口的list，通过nf_register_hooks来注册。具体注册过程不再赘述。

```
nf_register_hooks(ipv4_conntrack_ops,       ARRAY_SIZE(ipv4_conntrack_ops));
static struct nf_hook_ops ipv4_conntrack_ops[] __read_mostly = {
    {
        .hook           = ipv4_conntrack_in,
        .owner          = THIS_MODULE,
        .pf         = PF_INET,
        .hooknum        = NF_INET_PRE_ROUTING,
        .priority  = NF_IP_PRI_CONNTRACK,
    },
…
}
```

2. 关键数据结构

接下来的两节，重点考察Netfilter如何为每个skb创建一条连接跟踪，并如何维护它。相应的代码主要是nf_conntrack_core.c、nf_conntrack_l3proto_l4.c等几个文件。

使用数据结构nf_conntrack_tuple记录一条流的五元组，称之为一个tuple。使用数据结构nf_conn记录一个连线，该数据结构至少需要记录original和reply两个方向的tuple。

struct nf_conn

tuplehash[2]，记录连线两个方向的tuple

status，结构体状态，两个tuple是否都记录在案？

timeout，该连线何时可以被老化？

counters，有多少个包多少字节通过该连线？linux的防火墙可能对这些感兴趣

ct_general，当前连线被多少个skb所使用？**引用计数**！

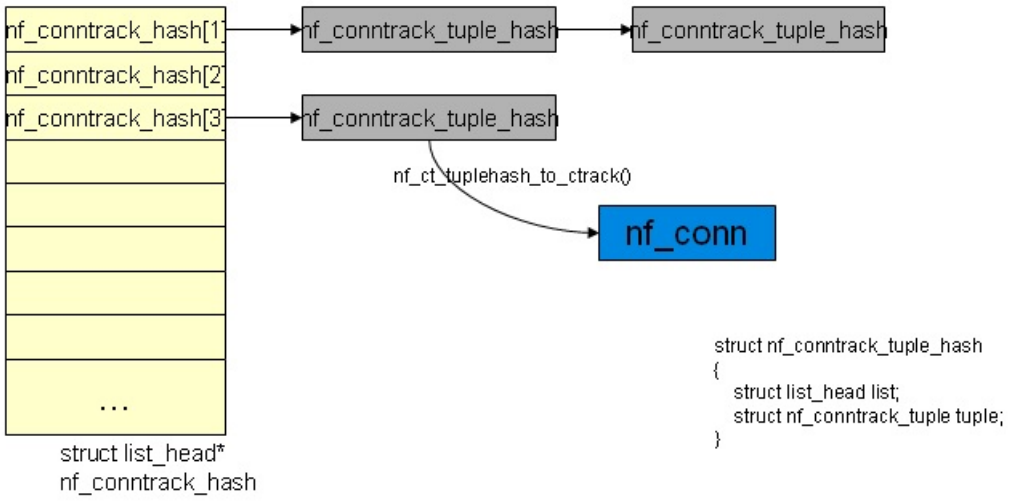
proto，该连线对应的协议，（好像和tuple冲突，但是很有必要）

struct nf_conntrack_tuple

src，五元组中源地址相关的，如srcIP，srcPort

dst，五元组中目的地址相关的，如dstIP，dstPort

由于modem需要保存多条流的连线数据，使用Hash表nf_conntrack_hash来存储众多的连线数据，以实现时间和空间的最优组合。Hash表大小\nf_conntrack_htable_size可以设置，hash表的键值通过五元组设置。



另外，在sk_buff结构体中有两个字段与连线跟踪有关：

struct sk_buff

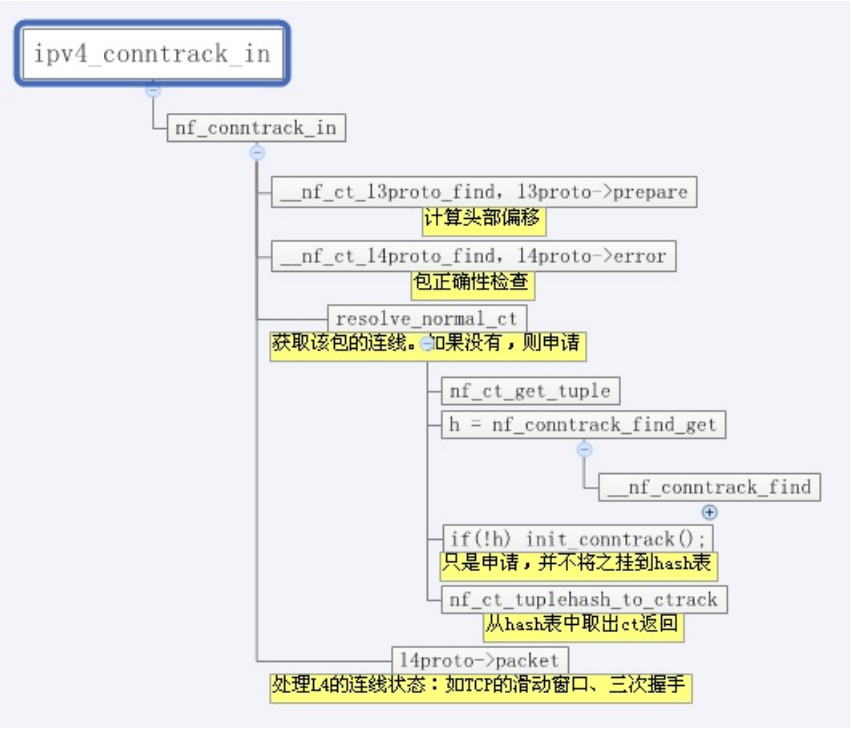
nfct 当前包是否被连线跟踪模块处理过？加速器根据该字段判断包是否进入L3处理

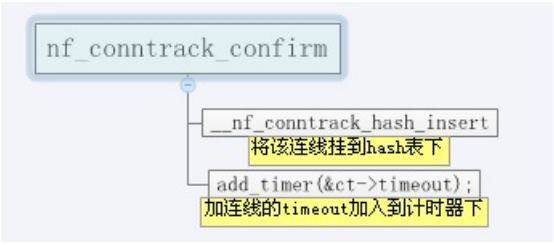
ctinfo 该包对应的连线的状态，如NEW、REPLY、ESTABLISHED等等

3. 关键过程

nf_conntrack_in，从NetFilter的PRE_ROUTING或者LOCAL_IN两个钩子点进入。

nf_conntrack_confirm，从NetFilter的POST_ROUTING或LOCAL_IN进入。





4. TCP协议

TCP状态管理

连线跟踪需要支持多种协议，如ICMP、TCP、FTP、SIP等，不同的协议对连线跟踪有一些特殊的要求，比如TCP协议，需要检查连线的TCP状态机，滑动窗口以判断包是否为非法包。

在上一层的流程图中看到，当包从NetFilter进入到连线跟踪模块，在nf_conntrack_in函数中，会调用l4proto->packet()函数，对TCP包做额外检查。L4proto是一个nf_conntrack_l4proto结构体，主要有以下字段：

```
struct nf_conntrack_l4proto
    u_int16_t l3proto;   //对应的l3协议
    u_int8_t l4proto;    //l4协议号
    const char *name;
    packet();          //流建立后，对每一个经过的包进行处理和检查
    new();             //新建连接时调用
    destroy();         //删除连接时调用
    …
```

调用nf_conntrack_l4proto_register()函数注册，所有的协议都存储在一个二维数组中。

连线跟踪和加速器

正常的一个包比如从lan口收到，或走二层桥转发或走三层路由通过pppoe或ipoe wan口发出去，这中间经过众多的协议栈处理，比如走三层路由的时候要经过iptables的各种表（连接跟踪，mangle表，filter表）逐条查找匹配，路由查找匹配，NAT转换等众多流程，这些都导致一个包处理时间很长，提高转发性能也就是减少一个包从一个口进入从另外一个口发出的时间。

一条流是由五元组（源IP,目的IP,源PORT,目的PORT,协议）决定的，对一条流来说，无论多少包经过协议栈这么多中间环节后的处理结果是一样的，比如NAT转换（发出去修改源，目的MAC,修改源IP,端口号等），比如从哪个端口发送出去（路由决定所有的包都是从同一端口发出去），比如从pppoe端口发送出去（pppoe端口发出去要在IP头前添加pppoe头，pppoe头里的session id 和以前的包是一样的）。可以看到对一条流，让第一个包经过这繁杂的协议栈中间环节处理后，将其处理结果记下来，则其后的包就不需要再通过这些中间环节了，直接用这些结果修改报文，从记录结果里指定端口发送出去就可以了。

加速器也要识别流，但加速器不仅把流识别出来，还要直接修改报文头部，然后调用发包设备将包发送出去。

CSP打算提供L3加速，是否就是这样实现的？修改连线跟踪模块，使得该模块在识别出一条流后，在模块内部记录。以后再有该流经过时，直接根据连线跟踪中的记录，修改报文头部，然后直接dst_output()出去？

在Modem中经常要解决一些连线跟踪的故障。特别地，当这些故障和加速器胶合在一起的时候就更难应付。因此需要对协议栈连线跟踪有充分的理解。本文从TCP状态机的角度来学习连线跟踪。

TCP需要通过三次握手创建连接，而关闭一条连接需要往来发送4个包。为深入了解TCP的状态机、定时器等，深入学习连线跟踪内部的状态图。client向server发TCP请求，报文经过Router被送到Server，Server回应该报文。定义orig和reply两个方向如下图所示。比如Router收到client发送的SYN报文，那么会创建一条连线跟踪，修改其状态为SYN_SENT。之后如果收到reply方向的SYN_ACK报文，那么TCP连接的状态修改为SYN_RECV。

连线跟踪中的TCP状态机

		sNO	sSS	sSR	sES	sFW	sCW	sLA	sTW	sCL	sLI	
orig	syn	sSS	sSS	sIG	sIG	sIG	sIG	sIG	sSS	sSS	sIV	1
	synack	sIV	sIV	sIV	sIV	sIV	sIV	sIV	sIV	sIV	sIV	2
	fin	sIV	sIV	sFW	sFW	sLA	sLA	sLA	sTW	sCL	sIV	3
	ack	sES	sIV	sES	sES	sCW	sCW	sTW	sTW	sCL	sIV	4
	rst	sIV	sCL	sCL	sCL	sCL	sCL	sCL	sCL	sCL	sIV	5
	none	sIV	sIV	sIV	sIV	sIV	sIV	sIV	sIV	sIV	sIV	6
reply	syn	sIV	sIV	sIV	sIV	sIV	sIV	sIV	sIV	sIV	sIV	7
	synack	sIV	sSR	sSR	sIG	sIG	sIG	sIG	sIG	sIG	sIV	8
	fin	sIV	sIV	sFW	sFW	sLA	sLA	sLA	sTW	sCL	sIV	9
	ack	sIV	sIG	sSR	sES	sCW	sCW	sTW	sTW	sCL	sIV	10
	rst	sIV	sCL	sCL	sCL	sCL	sCL	sCL	sCL	sCL	sIV	11
	none	sIV	sIV	sIV	sIV	sIV	sIV	sIV	sIV	sIV	sIV	12
		A	B	C	D	E	F	G	H	I	J	

```
enum tcp_conntrack {
    TCP_CONNTRACK_NONE,         //sNO
    TCP_CONNTRACK_SYN_SENT,     //sSS
    TCP_CONNTRACK_SYN_RECV,     //sSR
    TCP_CONNTRACK_ESTABLISHED,  //sES
    TCP_CONNTRACK_FIN_WAIT,     //sFW
    TCP_CONNTRACK_CLOSE_WAIT,   //sCW
    TCP_CONNTRACK_LAST_ACK,     //sLA
    TCP_CONNTRACK_TIME_WAIT,    //sTW
    TCP_CONNTRACK_CLOSE,        //sCL，进入CLOSE状态，包继续转发
    TCP_CONNTRACK_LISTEN,       //sLI
    TCP_CONNTRACK_MAX,          //sIV，丢弃包
    TCP_CONNTRACK_IGNORE        //sIG
};
```

上图中，syn、synack、fin、ack、rst、none表示连线跟踪模块收到一个包，其TCP标志位的类型。最上一行sNO、sSS等表示当前连线跟踪条目的状态。比如sNO表示连线跟踪条目尚未创建，sSS表示连线跟踪条目存在，其状态为SYN_SENT，即SYN报文已发送。

三次握手建立连接：A1->B8->C4。

A1一个新的SYN包进入到连线跟踪模块，（该包可能来源于中断，或者协议上下文），连接进入A1所示的状态，即sSS。同时设置该条目的老化时间为TCP_CONNTRACK_SYN_SENT，也就是2MINS。在2MINS之内如果没有收到回应报文，则将该连接老化掉。

B8 如果在此时间范围内，收到一个reply方向的SYNACK报文，修改连接状态为B8，连接进入SYN_RECV状态，同时设置老化时间。

B1 orig方向的SYN包，重传。

B8 Might be a half-open connection.

C4收到orig方向的ACK包，则连接进入sES状态。A1->B8->C4组成了三次握手创建TCP连接。之后就可以双向发送数据包了。数据包一般也是ACK报文。

A4 没有经过三次握手，直接进入sES状态

Router收到一个IP包，查找连线跟踪未命中，创建一条新的连接。检查TCP头部，该包是一个ACK包。没有经过三次握手，TCP连接尚未建立，收到ACK包是否非法并将该包丢弃呢？其实不然，在现网环境下，有多个路由设备存在。三次握手可能通过其他路由器完成，在client和server之间建立了连接。之后网络拓扑结构发生变化，ACK包经过一个linux设备转发，此时新建连接的状态为sES。

RST报文

第5行和第11行，在任何时候收到RST报文，连接都会进入到CLOSE状态。

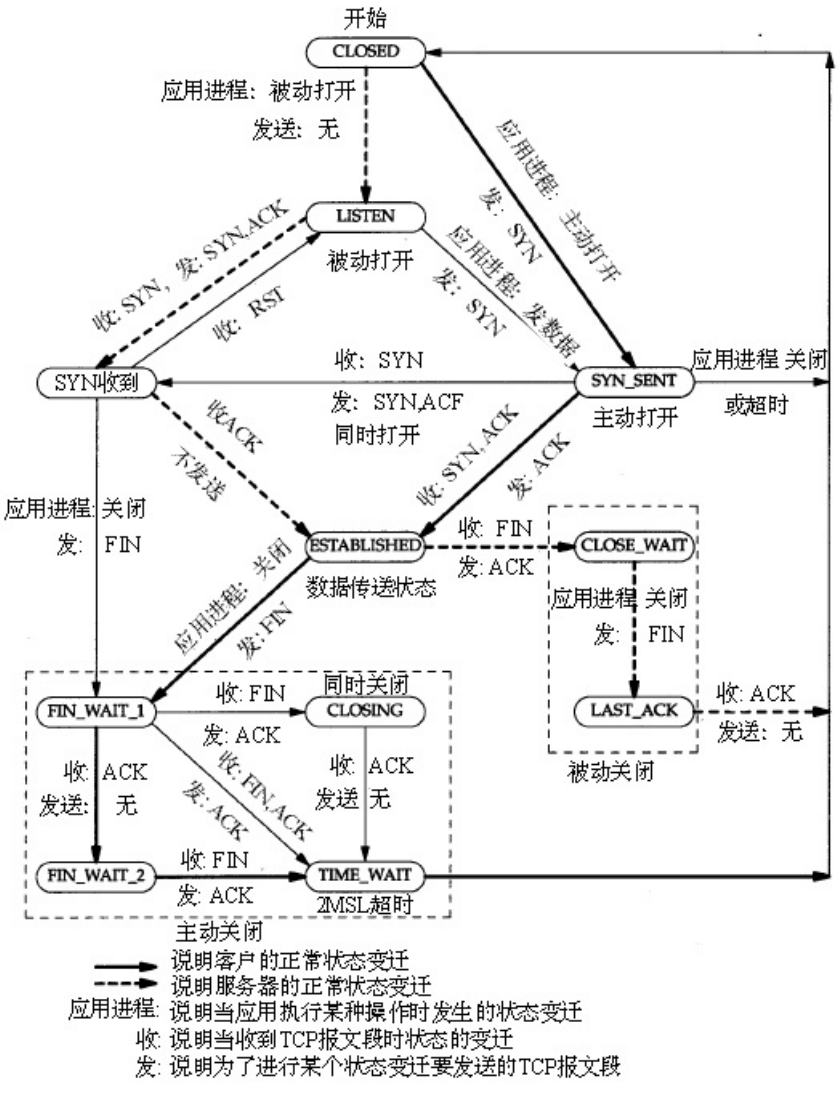
四次握手关闭连接：

D3->E10->F9->G4。orig方向发起关闭连接的操作。orig发FIN，reply方向回应ACK，reply方向回应FIN，orig方向回应ACK。orig方向会进入到TimeWait状态。

D3->E9->

TW状态的作用：连接关闭的时候有4次握手。某一方发送FIN包，向对端回应ACK包。之后对端会给它发送一个FIN包，它回应一个ACK包。此时发起方并不能直接进入CLOSE状态。因为如果回应给对端的ACK包在线路上丢掉了，对端还会超时重传一个FIN包过来，如果连接被关闭没有办法处理了。所以发起端在回应了对端的ACK报文后，还要等待2MSL的时间，防止对端重传FIN报文。

在开发网络程序的时候，由于编程故障，服务器端段错误退出。对于TCP连接而言，是服务器端发起的FIN，之后服务器会进入到TW状态。这会导致在2MSL时间内，服务器端口不可用。为避免这一问题，可以通过设置socket的SO_REUSEADDR来规避。只是规避，因为重用一个尚未关闭的端口是不符合TCP规范的。



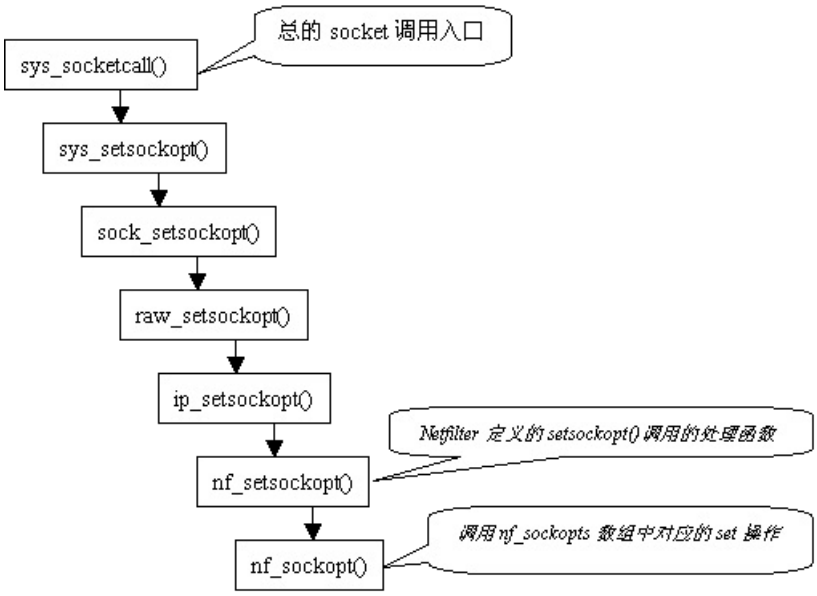
TCP状态

static unsigned int tcp_timeouts[TCP_CONNTRACK_MAX] __read_mostly = {

```
    [TCP_CONNTRACK_SYN_SENT]    = 2 MINS,
    [TCP_CONNTRACK_SYN_RECV]    = 60 SECS,
    [TCP_CONNTRACK_ESTABLISHED] = 5 DAYS,
    [TCP_CONNTRACK_FIN_WAIT]    = 2 MINS,
    [TCP_CONNTRACK_CLOSE_WAIT]  = 60 SECS,
    [TCP_CONNTRACK_LAST_ACK]    = 30 SECS,
    [TCP_CONNTRACK_TIME_WAIT]   = 2 MINS,
    [TCP_CONNTRACK_CLOSE]       = 10 SECS,
};
```

三. NAT

1. Iptables



对类似FTP这样的连接跟踪需要ALG协助的协议，它的包内容里包含了端口和IP信息，所以在做NAT转换的时候要将这些包内容也要做相应的转换，这些端口和IP信息的修改信息不是固定的，比如FTP 控制连接包内容里包含client打开的端口号，这个端口号一直在变，所以前一个包记录的修改值不能给后面的包用，所以明显这种包不能被加速，而且这种包通常是控制连接，流量不会很大。识别的方法很简单：struct ip_conntrack 里helper指针不空。

2. NAT
3. Mangle
4. Filter

四. ALG

附：连线跟踪相关的故障

931WII塞尔维亚版本私网地址泄漏

931WII塞尔维亚版本在WAN侧抓到来自192.168.1.2的包。

2. IPTables选项

IPTables提供了丰富的选项匹配数据包，也可以扩展已有的数据包。下面是1.2.11版的IPTables所提供的选项。

```
Usage: iptables -[AD] chain rule-specification [options]
       iptables -[RI] chain rulenum rule-specification [options]
       iptables -D chain rulenum [options]
       iptables -[LFZ] [chain] [options]
       iptables -[NX] chain
       iptables -E old-chain-name new-chain-name
       iptables -P chain target [options]
```

iptables -h (print this help information)

Commands:

Either long or short options are allowed.

```
  --append  -A chain           Append to chain
  --delete  -D chain           Delete matching rule from chain
  --delete  -D chain rulenum     Delete rule rulenum (1 = first) from chain
  --insert  -I chain [rulenum]     Insert in chain as rulenum (default 1=first)
  --replace -R chain rulenum      Replace rule rulenum (1 = first) in chain
  --list    -L [chain]           List the rules in a chain or all chains
  --flush   -F [chain]           Delete all rules in  chain or all chains
  --zero    -Z [chain]           Zero counters in chain or all chains
  --new     -N chain             Create a new user-defined chain
  --delete-chain -X [chain]          Delete a user-defined chain
  --policy  -P chain target        Change policy on chain to target
  --rename-chain  -E old-chain new-chain  Change chain name, (moving any references)
```

Options:

```
  --proto       -p [!] proto    protocol: by number or name, eg. `tcp'
  --source      -s [!] address[/mask]   source specification
  --destination -d [!] address[/mask]    destination specification
  --in-interface -i [!] input name[+]     network interface name ([+] for wildcard)
  --jump        -j target           target for rule (may load target extension)
  --match       -m match          extended match (may load extension)
  --numeric     -n                numeric output of addresses and ports
  --out-interface -o [!] output name[+]   network interface name ([+] for wildcard)
  --table       -t table            table to manipulate (default: `filter')
  --verbose     -v                verbose mode
  --line-numbers                  print line numbers when listing
  --exact       -x                expand numbers (display exact values)
[!] --fragment  -f                match second or further fragments only
  --modprobe=<command>          try to insert modules using this command
  --set-counters PKTS BYTES       set the counter during insert/append
[!] --version   -V                print package version.
```

3. 扩展IPTables

可以自定义ipt_match结构体，通过

可以在init_module()中通过ipt_register_match()函数向注册定义的ipt_match（2.6内核中的xt_match）结构体，实现自定义的规则。Xt_match是一个包含了名称，钩子，匹配函数、钩子、协议、销毁函数等部门的结构体。

```
struct xt_match
{
    struct list_head list;

    const char name[XT_FUNCTION_MAXNAMELEN-1];

    /* Return true or false: return FALSE and set *hotdrop = 1 to
       force immediate packet drop. */
    /* Arguments changed since 2.6.9, as this must now handle
       non-linear skb, using skb_header_pointer and
       skb_ip_make_writable. */
    int (*match)(const struct sk_buff *skb,
                 const struct net_device *in,
                 const struct net_device *out,
```

```
                const struct xt_match *match,
                const void *matchinfo,
                int offset,
                unsigned int protoff,
                int *hotdrop);

    /* Called when user tries to insert an entry of this type. */
    /* Should return true or false. */
    int (*checkentry)(const char *tablename,
                    const void *ip,
                    const struct xt_match *match,
                    void *matchinfo,
                    unsigned int hook_mask);

    /* Called when entry of this type deleted. */
    void (*destroy)(const struct xt_match *match, void *matchinfo);

    /* Called when userspace align differs from kernel space one */
    void (*compat_from_user)(void *dst, void *src);
    int (*compat_to_user)(void __user *dst, void *src);

    /* Set this to THIS_MODULE if you are a module, otherwise NULL */
    struct module *me;

    /* Free to use by each match */
    unsigned long data;

    char *table;
    unsigned int matchsize;
    unsigned int compatsize;
    unsigned int hooks;
    unsigned short proto;

    unsigned short family;
    u_int8_t revision;
};
```

参考：

http://www.ibm.com/developerworks/cn/linux/l-ntflt/