PLANETLAB

VNET: PlanetLab Virtualized Network Access

Mark Huang
Princeton University

PDN–05–029
June 2005

Status: Ongoing Draft.

VNET: PlanetLab Virtualized Network Access

Mark Huang

This document describes the design of VNET, the PlanetLab module that provides virtualized network access on PlanetLab nodes.

Table of Contents

Overview

This document describes the design of VNET, the PlanetLab module that provides virtualized network access on PlanetLab nodes. VNET replaces, but is also backward compatible with, the *safe raw sockets* interface supported by PLKMOD (also known as SILK) in version 2 of the PlanetLab software [1]. VNET provides a restricted form of raw IP and raw packet sockets, ensures isolation of traffic between slices, and supports a unique interface for accessing proxied IP addresses, all while maintaining compatibility with standard Linux/BSD socket APIs.

The goal of VNET is to be as transparent as possible. The great majority of PlanetLab users should not have to recompile any code, use anything besides standard programs and APIs, or even be aware that VNET virtualizes their network access. This document is intended to be read by PlanetLab users who require raw or close-to-raw access to the network.

Connection tracking

Connection tracking is the heart of VNET. VNET relies on Linux's Netfilter[1] system to associate every inbound and outbound IP packet with a connection structure. VNET then ensures that slices send and receive only packets associated with connections that they own. That is, slices can only:

- Send packets associated with new connections or connections that they initiated.
- Receive packets associated with connections that they initiated or bound.

When an IP packet is sent through a socket, it passes through VNET and is associated with a new or existing connection. If the connection is not already bound to a slice, VNET allows the packet through and binds the connection to the slice that sent the packet. If the connection is bound to a slice, and it is not the slice that sent the packet, the packet is dropped and an error is returned to the sending application.

When an IP packet is received by the stack, it also passes through VNET and is associated with a new or existing connection. If the packet was expected; that is, if the connection was bound by a slice; or the connection was initiated by a slice, VNET allows the slice to receive the packet. Otherwise, the packet can only be received by the root slice.

Connections

Connections are defined on a per-protocol basis. Currently, VNET supports the following protocols:

- TCP
- UDP
- ICMP
- GRE and PPTP

Associating packets with connections is not always trivial. For example, Netfilter considers some ICMP errors (e.g., most ICMP Port Unreachable messages) to be associated with the connection that caused the ICMP error to be generated, rather than the ICMP connection itself. A pleasant consequence of this behavior is that slices are automatically entitled to receive their own ICMP errors.

Connection reservation

Connections may be *bound*, or effectively reserved, by using the `bind(2)` system call. `bind(2)` is normally used on `SOCK_STREAM` or `SOCK_DGRAM` sockets to specify the interface and local TCP or UDP port number that a socket should use. For example, to create and bind a TCP socket that may be used to send packets from, or receive packets sent to, port 9090:

Example 1. Example C code for creating and binding a regular TCP socket.

```
int sock;
struct sockaddr_in sin;

/* Create a TCP socket */
sock = socket(PF_INET, SOCK_STREAM, 0);

/* Bind port 9090 on all interfaces to it */
memset(&sin, 0, sizeof(sin));
sin.sin_addr = htonl(INADDR_ANY);
sin.sin_port = htons(9090);
bind(sock, (struct sockaddr *) &sin, sizeof(sin));
```

Once a local port is successfully bound by a slice, no other slice may send or receive packets associated with that port. If another slice attempts to `bind()` the same port, the standard error `-EADDRINUSE` is returned. The binding slice may send or receive packets associated with the port through the bound socket (or through any other, typically raw, socket), for as long as the port is bound.

VNET and Netfilter thus effectively serve as a *stateful switching firewall* for local sockets. The system is stateful because connection state is continually tracked; it is switching because slices only see their own traffic; and it is a firewall because incoming traffic is redirected to the appropriate slice.

VNET extensions

VNET extends the Linux `socket(7)` interface for backward compatibility with PLKMOD, and to support reservation for IP protocols besides TCP and UDP. Additionally, VNET supports the use of Linux `packet(7)` sockets for compatibility with programs that expect full administrative control over network interfaces, such as **tcpdump**, `Alpine`[2], `Honeyd`[3], and `Click`[4].

Restrictions on sent packets

As previously mentioned, the primary restriction on sent packets is that slices may only send packets associated with connections that they own (i.e., new connections or connections that they initiated). Slices may send packets through any number of raw or regular sockets, although it is recommended that only a single raw IP or packet socket be open at any one time for the highest performance.

Because raw IP and packet sockets are generally used only by administrative programs, there are very few restrictions in the kernel stack on what can be sent through them. VNET thus implements its own restrictions on sent packets, and rejects malformed or otherwise disallowed packets with the standard error `-EPERM`, if any of the following conditions are met by a sent packet:

- The packet is malformed. In particular, runt or headerless packets or packets with non-local source IP addresses are not allowed. If a packet is sent through a raw IP socket that has had the `IP_HDRINCL` option set, and the source IP address field is left blank, it will be filled in by the kernel as usual.

- The packet is otherwise untrackable by Netfilter; in particular, IP multicast packets are not allowed.
- The connection with which the packet is associated, is owned by another slice.
- The packet is an ICMP error message. Only the kernel itself is allowed to generate ICMP errors. The ICMP messages that may be sent are:
  - Echo Request
  - Echo Reply
  - Timestamp
  - Timestamp Reply
  - Info Request
  - Info Reply
  - Address
  - Address Reply
- The TCP or UDP source port of the packet is below 1024. This port range is generally reserved for use by root services, but certain reserved ports may be bound via the Proper[5] [2] service.

Safe raw sockets

For backward compatibility with PLKMOD, and to support reservation for IP protocols besides TCP and UDP, `bind(2)` may also be used on `SOCK_RAW` sockets. To create a safe raw socket, the *`domain`* parameter of the `socket(2)` call must be set to `PF_INET`, the *`type`* parameter to `SOCK_RAW`, and the *`protocol`* parameter to `IPPROTO_TCP`, `IPPROTO_UDP`, `IPPROTO_ICMP`, or `IPPROTO_GRE` (the IP protocols supported by the PlanetLab Netfilter configuration). To bind an ICMP Echo connection, specify the ICMP Echo ID that will be used in the `sin_port` field of `struct sockaddr_in`. To bind a GRE or PPTP connection, specify the lower 16 bits of the 32-bit GRE key, or the 16-bit PPTP Call ID.

Example 2. Example C code for creating and binding a raw ICMP socket.

```
int sock;
struct sockaddr_in sin;

/* Create a raw ICMP socket */
sock = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);

/* Bind ICMP Echo ID 23456 to it */
memset(&sin, 0, sizeof(sin));
sin.sin_addr = htonl(INADDR_ANY);
sin.sin_port = htons(23456);
bind(sock, (struct sockaddr *) &sin, sizeof(sin));

/* Note that this socket will receive ALL ICMP packets that your slice
is entitled to receive. The act of binding simply ensures that your
slice will be able to send and receive packets associated with the
specified ICMP Echo ID, through this socket or others. */
```

Example 3. Example C code for creating and binding a raw GRE socket.

```
int sock;
struct sockaddr_in sin;

/* Create a raw GRE socket */
sock = socket(PF_INET, SOCK_RAW, IPPROTO_GRE);

/* Bind GRE key 12 (and/or PPTP Call ID 12) to it */
memset(&sin, 0, sizeof(sin));
sin.sin_addr = htonl(INADDR_ANY);
sin.sin_port = htons(12);
bind(sock, (struct sockaddr *) &sin, sizeof(sin));

/* Note that this socket will receive ALL GRE and PPTP packets that
your slice is entitled to receive. The act of binding simply ensures
that your slice will be able to send and receive packets associated
with the specified ICMP Echo ID, through this socket or
others. */
```

Raw IP sockets must be created as the root user. Most of the socket options described in the `ip(7)` **man** page, including `IP_HDRINCL`, affect socket behavior as documented, within the constraints listed in the Section called *Restrictions on sent packets*.

Suppression of kernel generated replies

Usually, the Linux kernel will respond to TCP or UDP requests to ports that it believes are non-listening, with a TCP RST packet or an ICMP Port Unreachable error. The kernel does not track the state of connections carried out through raw IP sockets, and can usually interfere with them.

VNET suppresses these kernel generated replies if a TCP or UDP port has been bound to a safe raw socket. This behavior is not standard and programs that rely on it, will not work on regular Linux.

Lazy binding

It is not absolutely necessary to bind local ports before using them. If a connection is not in use; that is, if the local source port of the connection has not been bound, and no other slice is sending packets from the same source tuple to the same destination tuple at the same time; then packets related to the connection may be sent and received through any number of regular or raw sockets until the port is bound by another slice.

Lazy binding is why stock **ping** and **traceroute** now work in version 3 of PlanetLab Linux without modification. Both of these programs use unbound raw IP sockets to send packets with effectively random ICMP Echo IDs. Each ICMP connection that these programs generate, is lazily bound to the slice running it.

Safe raw packet sockets

Linux `packet(7)` sockets may also be used to send and receive raw IP packets. Programs such as **tcpdump** and **snort** use packet sockets to capture and dissect IP traffic.

Example 4. Example C code for creating and binding a raw packet socket.

```
int sock;
struct ifreq ifr;
struct sockaddr_ll sll;
```

```
/* Open a generic IP socket for querying the stack */
sock = socket(PF_INET, SOCK_RAW, 0);

/* Get interface index */
memset(&ifr, 0, sizeof(ifr));
ifr.ifr_addr.sa_family = PF_INET;
strcpy(ifr.ifr_name, "vnet");
ioctl(sock, SIOCGIFINDEX, &ifr);

/* Re-open raw packet socket */
close(sock);
sock = socket(PF_PACKET, SOCK_DGRAM, htons(ETH_P_IP));

/* Bind packet socket to the "vnet" virtual Ethernet device */
memset(&sll, 0, sizeof(sll));
sll.sll_family = PF_PACKET;
sll.sll_protocol = htons(ETH_P_IP);
sll.sll_ifindex = ifr.ifr_ifindex;
bind(sock, (struct sockaddr *) &sll, sizeof(sll));
```

Standard `PF_PACKET` behavior and semantics are preserved, with three restrictions:

- Packet sockets must be bound to the vnet virtual Ethernet device before they can be used.
- The `sockaddr_ll` socket protocol must be `htons(ETH_P_IP)` or `htons(ETH_P_ALL)`. To support **tcpdump**, `htons(ETH_P_ALL)` is accepted but ignored; only IP packets may be sent or received through packet sockets.
- The socket *`type`* must be either `SOCK_DGRAM` or `SOCK_RAW`. If the socket is of *`type`* `SOCK_RAW`, then an Ethernet header must be included with each sent packet, but the source and destination MAC address fields may be empty or bogus. The Ethernet type field should be set to `htons(ETH_P_IP)`. If the socket is of *`type`* `SOCK_DGRAM`, then the `sockaddr_ll` destination MAC address to which the socket is bound may be empty or bogus. The `sockaddr_ll` protocol field should be set to `htons(ETH_P_IP)`.

VNET restricts the types of packets that may be sent through packet sockets. Normally, Linux packet sockets can be used to send any type of packet that the underlying device driver accepts. However, packets sent through VNET packet sockets are actually re-routed through the IP stack as if they were locally generated by a regular socket. Thus, VNET packet sockets may only be used to send well-formed, routable IP packets that follow the restrictions listed in the Section called *Restrictions on sent packets*.

Packet sockets may also be used to receive raw IP packets. IP packets are routed to the vnet device *after* they have been defragmented by the stack. The MRU of the vnet device may thus appear to be exceeded at times, but most standard programs such as **tcpdump** do not appear to care. As usual, slices may only receive packets related to connections that they own.

Proxy sockets

Proxy sockets are special packet sockets that allow slices to utilize unused IP address space on PlanetLab node subnets. On nodes with access to such address space, slices may bind to a virtual interface that proxies for the space. The interfaces should be treated as if they were anonymous (i.e., unconfigured for IP) Ethernet interfaces in promiscuous mode, connected to a subnet servicing the unused address space.

The primary applications for the interface are network telescopes and honey farms. Because proxy sockets do not require any special configuration, stock programs like **tcpdump**, `Honeyd`[6], and `Snort`[7] may be used to implement such applications. Other possible applications include user-level IP routers such as `Click`[8] and implementations of IP anycast.

The same restrictions that apply to safe raw IP and packet sockets, apply to proxy sockets as well. However, proxy sockets should be bound to an available proxy device of the form **proxy0**, **proxy1**, etc., rather than the vnet device. There is usually only one proxy device available per node at sites with available address space, and only one slice at a time is allowed to `bind()` to the device.

Packets sent through proxy sockets are re-routed through the IP stack as if they were forwarded from a foreign host. Thus, the normal restrictions listed in the Section called *Restrictions on sent packets* do not apply. As long as the packets are well-formed enough for the kernel to forward, they will be accepted.

Administrative interfaces

/proc/scout/ports/summary

VNET provides a version of `/proc/scout/ports/summary` that is backward compatible with PLKMOD's summary file[9]:

Example 5. Example listing of `/proc/scout/ports/summary`.

```
prot port slice types
tcp 52906 524 C
udp 9999 0 c
icmp 55170 874 c
tcp 49943 769 c
tcp 43693 769 c
udp 55196 874 c
tcp 39261 769 c
tcp 49075 769 c
udp 9999 0 c
```

The `I`, `R`, and `S` flags are not applicable to VNET. The meanings of the other flags are the same:

- [C]onsumer: This means the slice has bound a socket (either a standard TCP/UDP socket or a safe raw socket) on this port to all interfaces (`INADDR_ANY`).
- [c]onsumer: This means the slice has bound a socket (either a standard TCP/UDP socket or a safe raw socket) on this port to a (presumably external) non-loopback IP address.
- [l]oopback: This means the slice has bound a socket (either a standard TCP/UDP socket or a safe raw socket) on this port to the loopback IP address.

Lazy binds, discussed in the Section called *Lazy binding*, are also printed in `/proc/scout/ports/summary`. By definition, lazily bound sockets cannot be listening sockets and thus show up as either `c` or `l` type sockets.

/proc/net/ip_conntrack

Netfilter's `/proc/net/ip_conntrack` interface lists the state of all active connections:

Example 6. Example listing of `/proc/net/ip_conntrack`.

```
tcp      6 107 TIME_WAIT src=24.154.102.20 dst=128.112.139.71 sport=2550 dport=3127 xid
src=128.112.139.71 dst=24.154.102.20 sport=3127 dport=2550 xid=524 [ASSURED] use=1
icmp     1 29 src=128.112.139.71 dst=198.78.49.61 type=8 code=0 id=32111 xid=661 [UNREP
src=198.78.49.61 dst=128.112.139.71 type=0 code=0 id=32111 xid=-1 use=1
tcp      6 431583 ESTABLISHED src=128.238.35.25 dst=128.112.139.71 sport=41955 dport=31
src=128.112.139.71 dst=128.238.35.25 sport=3124 dport=41955 xid=524 [ASSURED] use=1
tcp      6 25 TIME_WAIT src=127.0.0.1 dst=127.0.0.1 sport=59101 dport=3100 xid=619 \
src=127.0.0.1 dst=127.0.0.1 sport=3100 dport=59101 xid=687 [ASSURED] use=1
udp      17 135 src=128.112.139.71 dst=140.112.107.82 sport=4121 dport=4121 xid=524 \
src=140.112.107.82 dst=128.112.139.71 sport=4121 dport=4121 xid=-1 [ASSURED] use=1
```

On PlanetLab, the listing is supplemented with information about the owner of each side of the connection (loopback connections may have two different owners). An `xid` value of `-1` means that the owner is unknown, or is external to the node.

The meaning of the other fields is the same. Examining the first and second entries in the example above:

1. `tcp 6`: The IP protocol of the connection.
2. `107`: The time left in seconds before the connection is timed out and considered expired. The timer is reset (to a constant value specific to each protocol) every time a packet related to the connection is received.
3. `TIME_WAIT`: (TCP only) The state of the TCP connection.
4. `src=24.154.102.20 dst=128.112.139.71`: The source and destination IP addresses of the connection in the original (i.e., initiated) direction.
5. `sport=2550 dport=3127`: (TCP and UDP only) The source and destination ports of the connection in the original (i.e., initiated) direction.
6. `type=8 code=0 id=32111`: (ICMP only) The ICMP type, code, and Echo ID fields of each packet of the connection.
7. `xid=-1`: The owner of the original end of the connection (i.e., the slice which initiated the connection). A value of `-1` means that the owner is unknown, or is external to the node.
8. `src=128.112.139.71 dst=24.154.102.20`: The source and destination IP addresses of the connection in the reply direction. These addresses may be different if Netfilter has been configured to perform some variety of NAT on packets. PlanetLab does not support NAT.
9. `sport=3127 dport=2550`: (TCP and UDP only) The source and destination ports of the connection in the reply direction. These ports may be different if Netfilter has been configured to perform some variety of NAPT on packets. PlanetLab does not support NAPT.
10. `xid=524`: The owner of the reply end of the connection in the original (i.e., initiated) direction. A value of `-1` means that the owner is unknown, or is external to the node.
11. `[UNREPLIED]`: This flag is protocol specific but generally means that no reply packets related to the connection have yet been seen.
12. `[ASSURED]`: This flag is protocol specific but generally means that reply packets related to the connection have been seen, and by some other heuristic, the connection is definitely active.
13. `use=1`: An internal counter used by Netfilter.

VNET socket options

VNET extends the meaning of the standard socket option `SO_PEERCRED` in two different ways. When the option is retrieved via `getsockopt()`, the `gid` field of `struct ucred` is overridden with the ID of the *slice* on the other end of the localhost socket, rather than the GID of the user within the slice:

Example 7. Example C code for retrieving the slice owner of the other end of a connection.

```
int sock;
struct sockaddr_in peer;
int peerlen = sizeof(peer);
struct ucred peercred;
socklen_t peercredlen = sizeof(peercred);

/* At this point, sock is connected to a peer. */

getpeername(sock, (struct sockaddr_in *) &peer, &peerlen);
getsockopt(sock, SOL_SOCKET, SO_PEERCRED, &peercred, &peercredlen);
printf("Peered with slice ID %d\n", peercred.gid);
```

When the option is set via `setsockopt()` (usually an unsupported operation), the ownership of the socket is *transferred* to the specified slice. Only the root slice is allowed to transfer socket ownership.

Example 8. Example C code for transferring ownership of a socket to another slice.

```
#if !defined(SO_SETXID)
#define SO_SETXID SO_PEERCRED
#endif

/* At this point, sock has been created and xid contains the ID of the
desired slice. */

setsockopt(sock, SOL_SOCKET, SO_SETXID, &xid, sizeof(xid));
```

Because setting `SO_PEERCRED` is usually an unsupported operation, as well as for clarity, it is recommended that `SO_SETXID` be used (and defined as `SO_PEERCRED` if necessary) instead.

Packet tags

VNET adds an `xid` field to `struct sk_buff` which stores the slice (or conteXt) ID of the "owner" of each packet. So that various standard programs do not have to be recompiled, this value is also copied to the `nfmark` field.

The `ipt_ULOG` module, in particular, can be used to pass out-of-band information about packets to userspace programs such as **ulogd**. The `nfmark` field of each packet is copied to the `mark` field of `ulog_packet_msg_t` and can be accessed within **ulogd** extensions by expressing an interest in `"oob.mark"`.

A. Examples

The following examples demonstrate the use of the VNET extensions described in this document. The code is freely redistributable within the constraints of the BSD license.

1. `rawping.c`[1]: A toy **ping** program that uses either raw IP or packet sockets to send and receive ICMP Echo packets. Demonstrates the use of `bind()` and packet sockets.
2. `ttcp.c`[2]: A modified version of **ttcp** that prints the owner of the other side of localhost connections. Demonstrates the use of `SO_PEERCRED`.

Bibliography

[1] PlanetLab Team, *Scout Module API*[1].

[2] Steve Muir, Marc Fiuczynski, Larry Peterson, Justin Cappos, John Hartman, *Proper: Privileged Operations in a Virtualised System Environment*[2], PDN-04-022, August 2004 (updated October 2004).

Notes

1. http://www.netfilter.org/
2. http://alpine.cs.washington.edu/
3. http://www.honeyd.org/
4. http://pdos.csail.mit.edu/click/
5. https://wiki.planet-lab.org/bin/view/Planetlab/ProperApi
6. http://www.honeyd.org/
7. http://www.snort.org/
8. http://pdos.csail.mit.edu/click/
9. http://www.planet-lab.org/raw_sockets/ports_summary.html

1. http://cvs.planet-lab.org/cvs/~checkout~/vnet/examples/rawping.c
2. http://cvs.planet-lab.org/cvs/~checkout~/vnet/examples/ttcp.c

1. http://www.planet-lab.org/raw_sockets/
2. http://www.planet-lab.org/PDN/PDN-04-022