XNetEm Network Emulation using XDP

Stephen Hemminger
Microsoft Corporation
`stephen@networkplumber.org`

Abstract

Many protocols and applications perform poorly when exposed to real life networks with delay and packet loss. Often, it is costly and difficult to reproduce Internet behavior in a controlled environment. There are tools available for testing, but they are either expensive hardware solutions, proprietary software, or one-off-research projects.

XNetEm is an update to NetEm using eXpress Data Path (XDP) to provide packet loss, corruption and marking at high data rates. Later versions will provide rate and latency impairments.

Keywords

XDP, eBPF, Linux, Network Emulation

Introduction

Network emulation is used as a testing tool for network protocols and applications. NetEm [7] is a network emulator built on the Linux Traffic Control (TC) subsystem. It has been used to emulate WiFi, satellite aircraft network, wide area network gaming [16], Next Generation Networks [4], and TCP protocol variants [18]. NetEm was even used in low end commercial network emulators [10]. NetEm is useful but is limited in the ability to handle high speed networks which lead to an investigation of possible alternatives.

This paper describes a proof of concept using the. Linux eXpress Data Path (XDP) [8] an infrastructure which provides low latency, high performance packet processing. XDP has already been used to provide high performance distributed Denial Of Service protection [1] and packet forwarding at full speed on 40 Gbit networks. NetEm is a natural type of application for XDP since it is small (a few hundred lines) and does not have any loops in the packet flow.

History

The first version of NetEm arose from the need to test new versions of TCP such as CUBIC [6] in a lab environment. There were several test tools such as NISTnet [2] and DummyNet [13] but they would not work for testing on the latest 2.6 kernel. DummyNet only worked on FreeBSD, and NISTnet was no longer supported (and required patches to Linux that were too invasive to upstream). The NetEm Queuing Discipline (qdisc) based solution was developed and incorporated into the standard Linux kernel. The original version of NetEm supported a full range of network impairments (and bugs), and these have been expanded and improved by the Linux network developers and research community.

Design

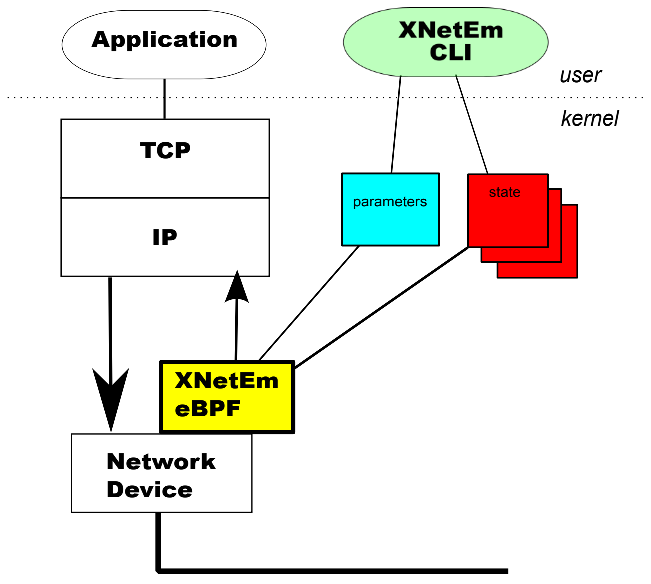

Figure 1: XNetEm architecture

The architecture of XNetEm is similar to other XDP applications. There is a user level process the CLI and a inserted kernel plugin to handle packets as shown in Figure 1. The kernel plugin is a C program that is compiled into extended Berkeley Packet Filter (eBPF) a special purpose virtual instruction set. This eBPF code is then evaluated by the kernel via network device (or generic network) XDP processing hook when packets are received. This eBPF code is automatically loaded by command line as needed.

The configuration is done by the user space program using arguments similar to the existing iproute2 **tc** qdisc [12]. The difference is that *tc* command converts these parameters into messages sent via netlink[14]; and the **xnetem** program converts these into eBPF map arrays.

NetEm syntax:

```
# tc qdisc add dev eth0 root netem \
    loss random 0.03 .25
```

XNetEm syntax:

```
# xnetem eth0 loss random 0.03 .25
```

Two eBPF maps are used: one is the parameter settings and the other is the current state of the program. The state map has statistics and other values that carry over from packet to packet.

In NetEm, all the network impairments are contained in one kernel module. For XNetEm, the individual emulation functions are broken up into different programs. This keeps each function smaller, and should improve performance.

Packet Loss

Packet loss is the easiest function to implement in XDP. It uses the basic `XDP_PASS` or `XDP_DROP` action offered by XDP. XNetEm supports the same rich models of packet loss as NetEm.

Random Loss

The simplest network impairment is random packet loss as shown in Listing 1.

The command line converts the probability in human readable format into a number scaled from 0 to the maximum value of a 32 bit unsigned integer and places that in the array shared between userspace and the eBPF program. For example: 1% becomes $42949672/$. The eBPF program then reads that value and compares it against a random value it obtains from `bpf_get_prandom_u32()`.

The actual program includes more parameters to handle correlated random generation. The original correlated random loss model in NetEm did not work as expected [9] both because of pure random is a poor model, and math errors in NetEm. Several correction parameters in NISTnet [2] are missing[1] causing unexpected results. These deficiencies in the original NetEm code, lead to a series of enhancements to NetEm to support alternative models [15].

Gilbert-Elliot loss model

A better loss model is the 2-state Markov model introduced by Gilbert [5] and Elliot [3] which is often used in network research.

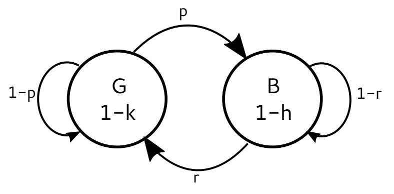

Figure 2: Gilbert-Elliott model of packet loss

[1]This will be addressed in XNetEm

This model uses two states, Good and Bad, and four parameters (p, r, h, $1-k$) to determine the probability of losing a packet in each state, and the probability of transitioning to the other state as shown in Figure 2.

XNetEm gemodel syntax:

```
# xnetem eth0 loss gemodel 20% 30% 70%
```

In this example the default (zero) is used for $1-k$.

The existing code from NetEm was converted from kernel to XDP for use in XNetEm as shown in Listing 2.

General and Intuitive Loss model

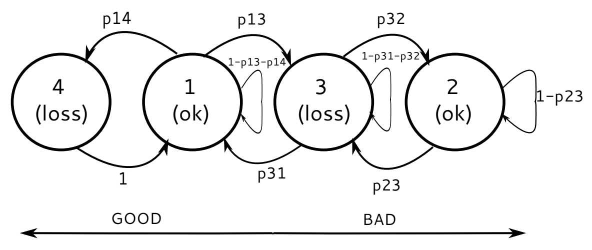

Figure 3: 4-state Markov model

1. packet received successfully
2. packet received with a burst
3. packet lost within a burst
4. isolated packet lost within a gap

The alternative model uses a 4-state Markov model in Figure 3. This model emulates both burst and gap periods to better describe how real network loss events happen. Most packets are lost either to burst noise or congestion in the network path, but there can also be sporadic one time losses due to random events such as collisions in shared media.

The transitions between states are controlled by a set of probabilities which are inputs to the model as shown in Listing 3. These probabilities are derived from the "General and Intuitive" model which uses:

- Loss probability P
- Mean burst length $E(B)$
- Loss density with burst ρ
- Isolated loss probability P_{ISOL}
- Mean good burst length $E(GB)$

The XNetEm command line takes care of converting the probability values into the units, state transitions and defaults.

XNetEm four state example

```
# xnetem eth0 loss state 20% 50%
```

This will get converted into to the state transitions with defaults applied of:

$p13$	20%
$p31$	50%
$p23$	1 (default)
$p14$	0 (default)

```
int xdp_loss_random_prog(struct xdp_md *ctx)
{
  int key = 0;
  u32 *val = bpf_map_lookup_elem(&options, &key);
  u32 prob = val ? *val : 0;
  int rc;

  if (prob >= bpf_get_prandom_u32())
    rc = XDP_DROP;
  else
    rc = XDP_PASS;
  return rc;
}
```

Listing 1: Random packet loss example

```
struct clgstate { /* Gilbert-Elliot models */
  u32 a1;         /* p for GE */
  u32 a2;         /* r for GE */
  u32 a3;         /* h for GE */
  u32 a4;         /* 1-k for GE */
};

static inline bool loss_event(const struct clgstate *clg,
                              unsigned long *state)
{
  u32 rnd1 = bpf_get_prandom_u32();
  u32 rnd2 = bpf_get_prandom_u32();

  switch (*state) {
  case GOOD_STATE:
    if (rnd1 < clg->a1)
      *state = BAD_STATE;
    if (rnd2 < clg->a4)
      return true;
    break;
  case BAD_STATE:
    if (rnd1 < clg->a2)
      *state = GOOD_STATE;
    if (rnd2 > clg->a3)
      return true;
  }
  return false;
}
```

Listing 2: XDP Gilbert-Elliot model example

```c
struct clgstate {         /* 4-states and Gilbert-Elliot models */
  u32 a1; /* p13 for 4-states or p for GE */
  u32 a2; /* p31 for 4-states or r for GE */
  u32 a3; /* p32 for 4-states or h for GE */
  u32 a4; /* p14 for 4-states or 1-k for GE */
  u32 a5; /* p23 used only in 4-states */
};

static inline bool loss_4state(const struct clgstate *clg,
                               unsigned long *state)
{
  u32 rnd = bpf_get_prandom_u32();

  switch (*state) {
  case TX_IN_GAP_PERIOD:
    if (rnd < clg->a4) {
      *state = LOST_IN_BURST_PERIOD;
      return true;
    }
    if (clg->a4 < rnd &&
        rnd < clg->a1 + clg->a4) {
      *state = LOST_IN_GAP_PERIOD;
      return true;
    }
    if (clg->a1 + clg->a4 < rnd) {
      *state = TX_IN_GAP_PERIOD;
    }
    break;
  case TX_IN_BURST_PERIOD:
    if (rnd < clg->a5) {
      *state = LOST_IN_GAP_PERIOD;
      return true;
    } else {
      *state = TX_IN_BURST_PERIOD;
    }

    break;
  case LOST_IN_GAP_PERIOD:
    if (rnd < clg->a3)
      *state = TX_IN_BURST_PERIOD;
    else if (clg->a3 < rnd &&
             rnd < clg->a2 + clg->a3) {
      *state = TX_IN_GAP_PERIOD;
    } else if (clg->a2 + clg->a3 < rnd) {
      *state = LOST_IN_GAP_PERIOD;
      return true;
    }
    break;
  case LOST_IN_BURST_PERIOD:
    *state = TX_IN_GAP_PERIOD;
    break;
  }
  return false;
}
```

Listing 3: XDP Generalized Intuitive model example

Packet Corruption

Random packet corruption with XDP is almost as simple to implement as packet loss as show in Listing 4. Two random numbers are used: the first determines the probability that a packet will be modified, and the second determines which bit in the packet will be flipped.

```
int xdp_corrupt_prog(struct xdp_md *ctx)
{
  int key = 0;
  u32 *val = bpf_map_lookup_elem(&options, &key);
  u32 prob = val ? *val : 0;
  int rc;

  if (prob >= bpf_get_prandom_u32()) {
    u32 pktbits = (data_end - data) * 8;
    u32 pos = bpf_get_prandom_u32() % pktbits;

    data[pos >> 3] ^= 1u << (pos & 7);
  }
  return XDP_PASS;
}
```

Listing 4: XDP Random Corruption program

Limitations

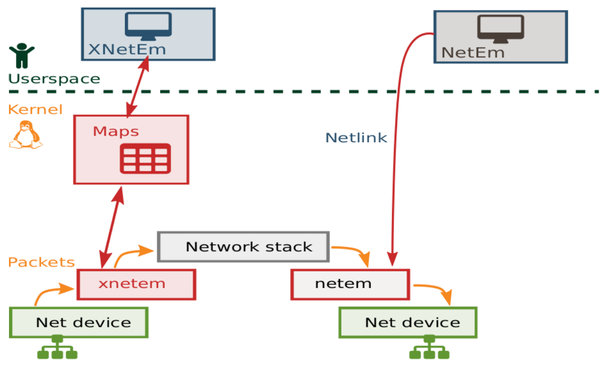

Figure 4: XNetEm vs NetEm

XNetEm has several limitations which prevent implementing some of the network impairment functions present in NetEm. Since XDP is normally done immediately on packet reception, XNetEm happens at a different point in the Linux packet processing stack as shown in Figure 4. Because XNetEm runs at such a low level it is able to process more packets per second but it is limited in functionality.

Queuing Disciplines like NetEm are designed to allow for traffic shaping where bursts of packets are spread in time to keep traffic above an allowed maximum. Internally, queue disciplines have a queue which is usually First In First Out (FIFO). NetEm uses its internal queue to apply impairments such as added delay, reordering and rate control. Since it is native kernel code NetEm can use other facilities such as high speed clocks and timers which allows for more features.

XDP applications are more targeted at firewall and policing type services. Applications such as XNetEm can not queue packets, they can only give an action verdict such as DROP, PASS or REDIRECT. There is also no way to invoke an XDP program as the result of a timer event. These limitations are why XNetEm can not easily implement the delay, reorder or rate control impairments.

Conclusion and Future Directions

The current prototype code has limited functionality and has not yet been fully tested. The first test will be to measure the accuracy of the loss model under packet load of 14.8 Million packets/sec (pps) or more. The earlier investigations of NetEm showed that it is limited to about 1 Mpps.

The first planned enhancement is to allow impairments to be cascaded using the chaining feature of XDP. This will allow combining packet loss and packet corruption.

The current packet loss logic can also be enhanced to support Explicit Congestion Notification (ECN). If this option is set then instead of returning `XDP_DROP` the loss will be converted into a modification of the IP packet header to set ECN bit.

It should be possible to modify NetEm GUI environments such as NetShaper project [17] to support XNetEm. This would make the tool more accessible for developers unfamiliar with XDP.

Rate control via policing is also possible by measuring the packet arrival time using (or enhancing) XDP clock functions. There is an implementation of Token Bucket Filter(TBF) in BPF already[11].

The major missing functionality in XNetEm is the ability to delay packets. The delay function also makes reordering possible. The problem is that current XDP architecture does not have enough functionality to implement packet holding and timers.

There are several possible ways to implement this:

- enhance XDP to allow deferring and injecting packets at a later time; or
- use *tc* classifier eBPF, instead of at XDP (this would allow for queuing but at the expense of performance); or
- using hardware timestamping and pacing to emulate delay and reordering.

With these enhancements, XNetEm will allow Linux to continue to be used for research in high speed network protocols and environments.

Acknowledgments

XNetEm is a promising new direction for high performance network emulation but is built on foundations provided by a wider community. Without the patches and papers it would not be possible to build such this kind of tool.

References

[1] Bertin, G. 2017. Xdp in practice: integrating xdp into our ddos mitigation pipeline. *Netdev 2.1*.

[2] Carson, M., and Santay, D. 2003. Nist net: a linux-based network emulation tool. *ACM SIGCOMM Computer Communication Review* 33(3):111–126.

[3] Elliott, E. O. 1963. Estimates of error rates for codes on burst-noise channels. *The Bell System Technical Journal* 42(5):1977–1997.

[4] Fabini, J.; Reichl, P.; Egger, C.; Happenhofer, M.; Hirschbichler, M.; and Wallentin, L. 2008. Generic access network emulation for ngn testbeds. In *Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities*, 43. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[5] Gilbert, E. N. 1960. Capacity of a burst-noise channel. *Bell Labs Technical Journal* 39(5):1253–1265.

[6] Ha, S.; Rhee, I.; and Xu, L. 2008. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review* 42(5):64–74.

[7] Hemminger, S., et al. 2005. Network emulation with netem. In *Linux conf au*, 18–23.

[8] Herbert, T., and Starovoitov, A. 2017. express data path (xdp).

[9] Jurgelionis, A.; Laulajainen, J.-P.; Hirvonen, M.; and Wang, A. I. 2011. An empirical study of netem network emulation functionalities. In *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, 1–6. IEEE.

[10] Labs, I., and Wellens, C. 2017. Netem with a gui.

[11] Monnet, Q. 2017. Stateful packet processing: two-color token-bucket poc in bpf.

[12] Pfeifer, H. P., and Ludovici, F. 2017. tc-netem(8) - linux manual page.

[13] Rizzo, L. 1997. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review* 27(1):31–41.

[14] Salim, J.; Khosravi, H.; Kleen, A.; and Kuznetsov, A. 2003. Linux netlink as an ip services protocol. Technical report.

[15] Salsano, S.; Ludovici, F.; Ordine, A.; and Giannuzzi, D. 2012. Definition of a general and intuitive loss model for packet networks and its implementation in the netem module in the linux kernel. *University of Rome «Tor Vergata». Version* 3.

[16] Tzruya, Y.; Shani, A.; Bellotti, F.; and Jurgelionis, A. 2006. Games@ large-a new platform for ubiquitous gaming and multimedia. *Proc. BBEurope, Geneva, Switzerland* 11–14.

[17] VanderLinden, J. 2017. Very simple ui for basic network traffic shaping using tc-netem.

[18] Yamamoto, T. 2008. Estimation of the advanced tcp/ip algorithms for long distance collaboration. *Fusion Engineering and Design* 83(2):516–519.