KauNetEm: Deterministic Network Emulation in Linux

Johan Garcia, Per Hurtig
Karlstad University, Karlstad, Sweden
{johan.garcia, per.hurtig}@kau.se

Abstract

This paper presents KauNetEm, an extension to the Linux-based NetEm emulator that provides deterministic network emulation. KauNetEm enables precise and repeatable placement of NetEm emulation effects, a functionality that can considerably simplify several aspects of protocol evaluation. KauNetEm can be instructed to drop specific packets, apply a configurable delay or other emulation effects at predefined points in time. The motivation for deterministic emulation, the overall design of KauNetEm, and usage examples are provided.

1 Introduction

When conducting network research and/or developing communication systems and mechanisms, ways of assessing the behavior and performance of various protocols are necessary. Often used methods include mathematical modeling and analysis, simulation, emulation and live experimentation. The different methods have their respective merits depending on what is to be assessed, and at which level of detail. For example, when evaluating the performance of proposed algorithms/mechanisms that are not yet supported by any implementation, analysis or simulation are the only possible choices. Furthermore, the choice of method is also an implicit choice of abstraction level. For instance, a simulation will capture idealized high-level characteristics while real experiments captures both the effects of the object under study as well as all its interactions with the surrounding environment.

Network emulation has for a long time been used to balance between these levels of abstraction. That is, emulation enables experimentation with real applications and protocols but controls the network characteristics. This is for example useful when analyzing real transport protocol implementations over a wide range of emulated network conditions. Although emulation can provide such balance, two critical problems often arise when emulating: (i) knowing exactly what is emulated; and (ii) being able to exactly reproduce an emulated scenario. For example, assume that we want to see how packet loss affects the performance of file transfers using the reliable transport protocol TCP. To conduct the experiments we set up an environment where the TCP flow is sent through a NetEm qdisc instructed to drop 1% of all packets. However, such loss specification adds an amount of ambiguity. It is not possible to know beforehand the exact amount of packets that will be lost, and when these losses occurs. TCP is known to be ineffective in recovering packets lost at specific positions in a flow, making it important to know and/or control which packets that are dropped. Moreover, if one wants to study the effect of e.g. TCP tail loss (a known problem when packets are lost at the end of a flow), it is impossible to construct such a scenario using only a drop percent parameter.

KauNetEm solves this problem by allowing an experimenter to precisely apply emulation effects, either on a per-packet basis or on a per-millisecond basis. We now consider the TCP tail loss problem. TCP has problems with recovering loss at specific positions in a flow, specifically in the beginning and the end of transfers, where packet loss causes long delays in the transmission. Therefore, losing one of the last packets in a TCP flow might result in increased delays. Using a regular non-deterministic network emulator, it would be hard to measure the effects of losing the last packet of a flow as there are no mechanisms to accomplish that situation. This is no problem using KauNetEm. Figure 1 shows the transmission times of 20 different TCP flows. The evaluation is done for two different settings of the net.ipv4.early_retrans variable for a TCP sender running Linux kernel 4.4.5. Each point represents the time required to transmit the entire flow, given

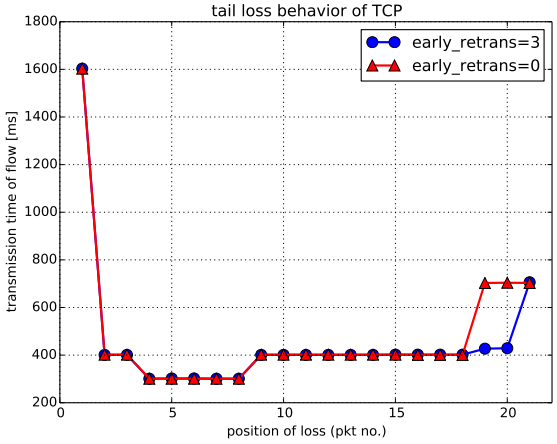

Figure 1: Transmission time of a 20-packet TCP flow experiencing a single packet loss at a certain position in the flow.

that a certain packet in the flow is lost (given by the x-axis). For instance, if the third packet is lost the transmission will take 400 ms while a loss of the last packet will require more than 650 ms. This kind of evaluation is simple to conduct in KauNetEm, it is only a matter of creating a packet-loss pattern that instructs the emulator to lose a specific packet, e.g., packet no. 3, and load that pattern into the emulator before starting the TCP flow. KauNetEm does not only provide this functionality for packet loss, but for all emulation effects originally supported by NetEm.

The remainder of this paper is structured as follows. Section 2 describes how to use KauNetEm through some easy usage examples. The design of KauNetEm is covered in Section 3, and the different emulation effects are described in more detail in Section 4. The paper is then concluded with a discussion on related works in Section 5 and some concluding remarks including possible future work in Section 6.

2 Usage examples

This section demonstrates how to use KauNetEm in a few simple examples. In the first example we instruct KauNetEm to lose the third, fifth and seventh packet of an ICMP flow headed towards Google. Then, we show how to apply emulation effects on selected traffic only. Listing 1 shows how to create a packet loss pattern. This particular pattern has the size of 20 packets, i.e., it can be used to control whether 20 packets should be lost or not. The pattern is named packet_loss.pkt and, when used, will cause packets three, five, and seven to be dropped. The last line of Listing 1 simply moves the pattern to the tc[1] load directory and renames it to tc's default file extension.

Listing 1: Creating a packet loss pattern

```
# Create a packet loss pattern of length 20 packets, that
# causes packets 3, 5 and 7 to be lost
$ patt_gen -pkt -s 20 -o packet_loss.pkt 3,5,7

# Move the packet to the tc dir and name it ploss.dist
# (.dist is the default extension)
$ mv packet_loss.pkt /usr/lib/tc/ploss.dist
```

When the pattern has been created and moved into the right location, it is time to configure KauNetEm to use it. Listing 2 shows how this is done. First, tc is used to attach a KauNetEm qdisc to the interface eth0. The qdisc is configured to have a data-driven[2] loss pattern, named ploss(.dist), attached to it. As seen above, we configured the loss pattern to drop packets three, five, and seven. Looking at the second part of Listing 2 we indeed see that the ICMP messages with the corresponding sequence numbers are lost.

Listing 2: Loading a packet loss pattern and applying it on ICMP traffic

```
# Create and configure a KauNetEm qdisc that uses the
# (data-driven loss pattern "ploss.dist")
$ tc qdisc add dev eth0 handle 1: root netem pattern data loss ploss

# Start experiment traffic
$ ping -c 10 www.google.com
PING www.google.com (216.58.213.100): 56(84) bytes of data.
64 bytes from 216.58.213.100: icmp_seq=1 ttl=63 time=16.095 ms
64 bytes from 216.58.213.100: icmp_seq=2 ttl=63 time=16.175 ms
64 bytes from 216.58.213.100: icmp_seq=4 ttl=63 time=14.145 ms
64 bytes from 216.58.213.100: icmp_seq=6 ttl=63 time=16.838 ms
64 bytes from 216.58.213.100: icmp_seq=8 ttl=63 time=17.106 ms
64 bytes from 216.58.213.100: icmp_seq=9 ttl=63 time=20.592 ms
64 bytes from 216.58.213.100: icmp_seq=10 ttl=63 time=18.956 ms

--- www.google.com ping statistics ---
10 packets transmitted, 7 packets received, 30% packet loss
rtt min/avg/max/mdev = 14.145/17.192/21.131/2.196 ms
```

When setting up experiments, care must be taken to isolate and only apply emulation effects on the experimental traffic. For instance, let us suppose that a routing or ARP message were sent over eth0 in the above example, and that this message was sent over the interface between ICMP messages with sequence numbers two and three. In that scenario, KauNetEm would have dropped the non-ICMP message instead of the ICMP with sequence number three. To avoid such situations it is possible to use the iptables utility to mark the traffic that is relevant for the experiment, and instruct KauNetEm to only apply effects to packets with the corresponding marking. Listing 3 shows an example of how this is accomplished. In the example, a rule is created to mark all ICMP packets with a certain number, in this case 42. Then, when the KauNetEm qdisc is created and configured we instruct it to only apply the emulation effects on traffic that has this exact marking. This KauNetEm-specific approach is a straightforward approach to filter out which traffic to perform emulation on, but other approaches are also possible. Using a nested queuing discipline structure is discussed in [6], and more details are also available in [8].

Listing 3: Filter specific traffic through KauNetEm

```
# Mark traffic to be experimented on with the number 42
$ iptables -A PREROUTING -t mangle -p icmp -j MARK \
           --set-mark 42

$ tc qdisc add dev eth0 root netem fwmark 42 \
     pattern data loss ploss
```

3 System Design Aspects

3.1 Design overview

A structural overview of the design is provided in Figure 2. When implementing the deterministic emulation, a goal was to minimize the amount of extra code necessary to achieve this functionality. By using the emulation effect code already implemented in NetEm, the added code mainly includes functionality to perform pattern handling and control value decoding. As NetEm has functionality to import distribution tables into the kernel at runtime, KauNetEm reuses parts of this functionality to transfer the patterns from user space into kernel space. This transfer is done by means of netlink sockets [14, 11]. The amount of data transferred into the kernel is, for efficiency reasons, kept as small as possible, so care is taken to ensure that the values for emulation effects are represented in an efficient manner as discussed in Section 4.

[1] Traffic Control (tc) is used to create, configure, and manage all Linux qdiscs.

[2] KauNetEm patterns are either data-driven or time-driven. When data-driven, effects are applied on per-packet basis and when time-driven effects are applied on per-millisecond basis as shown in Section 3.3.

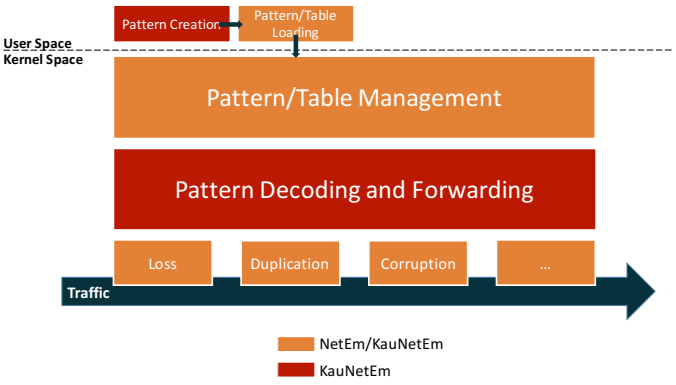

Figure 2: Structural overview of the design

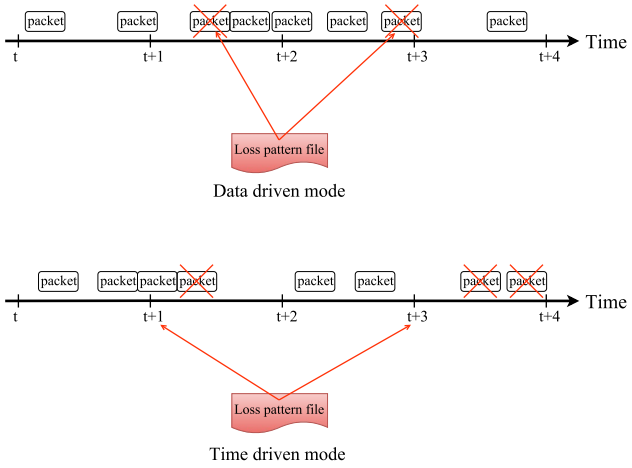

Figure 4: Data versus time driven mode

3.2 Pattern management

As mentioned above, existing functionality is reused to transfer pattern data from user space to kernel space. The emulation pattern control data is structured into the same appearance as a NetEm distribution file and should therefore be stored in the /usr/lib/tc/ directory expected by the tc distribution import mechanism. To create these distribution-like pattern control files, the patt_gen utility is used. The relationships among the different components are illustrated in Figure 3. The patt_gen utility can get the pattern specifications directly from the command line, or read them in from a comma-separated text file. Such text files could come from a variety of sources depending on the purpose of the emulation setup.

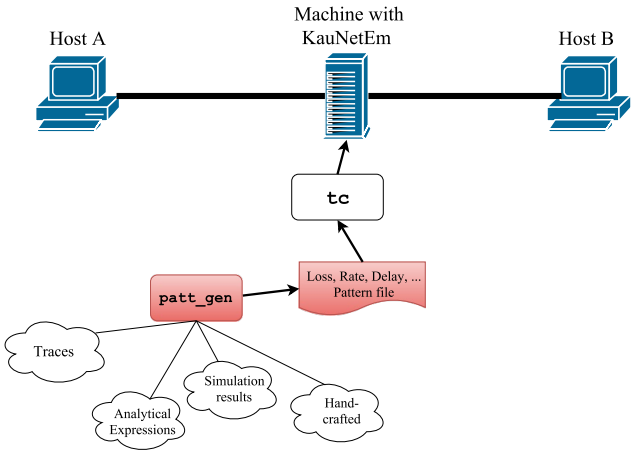

Figure 3: Pattern creation and import components

A possible future development is to integrate the relevant aspects of the patt_gen functionality into tc, which would remove the need of the patt_gen user space utility. Furthermore, handling of pattern files encoded as distribution files would no longer be necessary, as a text file based pattern specification would then be encoded by tc on the fly as patterns are moved into the kernel.

3.3 Time vs data driven

KauNetEm allows the emulation effects to be employed either in a data-driven or time-driven mode. For the data-driven mode, the movement in the emulation control pattern is done on a per-packet basis. For the time-driven mode, the pattern forwarding is instead performed at regular time steps, with the default time step being 1 ms. The two modes of pattern forwarding are illustrated in Figure 4.

For data-driven patterns it is, as previously mentioned, important to ensure that only the relevant traffic is passed through KauNetEm. If an examination is using precisely placed losses to evaluate the effect of some TCP mechanism, it is important that no other traffic such as routing messages or ARP is passing through the same KauNetEm qdisc as the TCP flow.

Time-driven pattern movement is done as a function of time and has as such no dependence on the traffic. For time-driven patterns, the only traffic-dependent aspect is the initial start. The timer used to forward time driven patterns are started when the first packet is observed by the qdisc where KauNetEm is located. The emulation effects are then applied at the time offsets specified in the pattern file, relative to the first observed packet.

4 Emulation effects

KauNetEm supports several emulation effects. An overview of the effects, and for which modes they apply, is shown in Table 1. The different emulation effects require different types of values to control them. Three different types of value representations are used in KauNetEm, and in the following subsections the emulation effects are discussed grouped according to the type of values they use for control.

4.1 Rate and Delay

Rate and delay patterns allow for precise control of changes to the emulated rate and delay. One use case where this could be useful is to represent the characteristics of a link where the link capacity varies over time, such as for wireless links. For data-driven patterns the value of the rate/delay is changed

Emulation effect	Data-driven	Time-driven
Packet loss	X	X
Delay	X	X
Rate	X	X
Bit error	X	-
Duplication	X	X
Reordering	X	-

Table 1: Emulation effects overview

when the indicated packet is received, and for time-driven the value is changed at the indicated time. The rate and delay values can vary over a wide range, and consequently the value encoding needs to be able to represent a large range. This large value range requirement opens up for a trade-off between the space needed to represent the patterns, and the precision with which values can be represented.

For rate and delay patterns an 11+4 floating point representation is used during the transfer into the kernel, which allows values in the range 1 to $2.048 * 10^{15}$ to be represented. Given the restriction that each value should fit within a 16 bit short, some constraints on precision are unavoidable. The level of precision in the representation is dependent on the particular value that should be stored. A graph showing the amount of representational error is shown in Figure 5. From the graph it

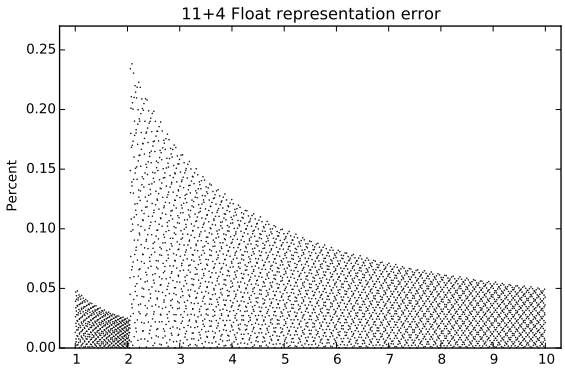

Figure 5: Value representation error

can be seen that the worst case error is less than 0.25%, and it occurs just after 2.048. A value of $2.1549 * 10^6$ bps will for example be stored as $2.15 * 10^6$ bps , and thus give a representational error of slightly less than 0.25%. Values with three or less value digits, such as $9.65 * 10^4$ etc. will always have zero representational error. When the values are decoded and used inside the kernel, they are represented as 64 bit integers.

4.2 Packet loss and Duplication

Packet loss and duplication patterns allows controlled placement of loss and duplicated packets. In data-driven mode, the patterns control the drop or duplication of individual packets in the received packet sequence. For data-driven mode, control is performed on a per-millisecond basis. For each millisecond that is indicated in a pattern, all packets that arrives during that millisecond are dropped/duplicated. As these patterns do not need to convey any values, they use a 15 bit run-length encoding to represent positions in the packet sequence/time.

The pattern creation functionality allows both the explicit placement of losses at particular positions, as well as creating patterns where each packet has a certain loss probability, or patterns where a specified number of losses should be placed in a pattern of a specific size. It is important to note that the use of a random loss probability, or alternatively a specific number of randomly placed losses, can lead to a considerable difference in result.

Consider a case where a TCP flow of 1000 packets should be evaluated over a link with 0.2% losses. Using regular stochastic emulation this means that each packet has a 0.2% probability of being lost, and that the placement of lost packets will be different from run to run. With deterministic emulation it is possible to create a pattern by specifying a 0.2% packet loss probability. Using such pattern will create random losses, but unlike the previous case the position of these losses can be reliably replicated.

However, it is also possible to specify the number of packets that should be lost. Consider again the 1000 packet flow which should be subjected to a 0.2% packet loss probability. 1000 packets with 0.2% loss should equate 2 lost packets. However, the binomial probability mass function (PMF) tells us that for 1000 draws at a 0.2% probability there is in fact a probability of 0.16 that no packets at all will be lost. Further, it is almost equally probable that one packet is lost instead of the expected two. The relevant PMF is shown on the left in Figure 6. By using the ability of the pattern gen-

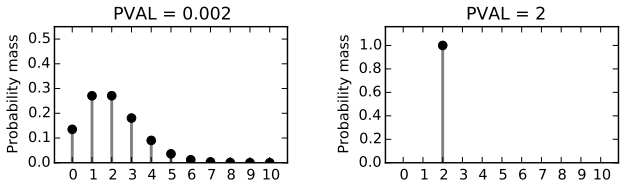

Figure 6: Distribution of number of packet losses for a 1000 packet flow when using 0.002 packet loss probability (left) or deterministic 2 packets loss specification (right)

erator to create a pattern with a size of 1000 packets and the required number of losses it is instead possible to create patterns where the losses are placed randomly, but which always has two losses. The resulting PMF is shown to the right in Figure 6. As transport protocols are typically very sensitive to the presence and placement of losses, it is clearly useful to be able to ensure that losses actually occurs, instead of including a fraction of runs where no actual losses occurred.

4.3 Reordering and Bit errors

Reordering and bit error patterns uses a value to signify the number of packets to reorder a given packet, and where the bit error should occur, respectively. Unlike the case for rate, the required range of values is quite small. A 15 bit integer

representation is used for these values. For reordering, the value signifies the number of later packets that should pass before the reordered packet is inserted back into the packet stream. For bit errors, the value signifies the position of the bit that should be flipped. The bit error patterns thus allows for the insertion of one single bit flip, at a specified position in a specific packet. In addition to these patterns, a planned trigger value emulation effect will also use an integer value to carry experiment specific trigger values to external listening applications. This triggering mechanism can for example be used to emulate various cross layer signals as done in [13].

5 Related Work

Network emulation has been in use, and been a topic of studies, for a long period. A general discussion of when emulation is a suitable approach is provided in [9]. In Linux, emulation functionality was early provided by NistNet [2]. In the current Linux kernel emulation functionality is provided by NetEm [6] and, as discussed in Section 3, it provides the framework extended by KauNetEm.

Network emulation is also available in other operating systems, and in FreeBSD DummyNet [1] provides network emulation functionality. DummyNet has also been made available for Linux and Windows. With regards to deterministic emulation, a DummyNet-based functionality has been developed earlier, under the name KauNet [4]. A discussion of the statistical benefits of using deterministic emulation for packet losses is provided in [3]. Deterministic emulation has also been further extended to allow state-based application of emulation effects, for example to emulate more complex interactions such as DVB-RCS resource allocation [5].

Different emulators have different design considerations and can differ in what emulation effects they can provide, and what precision they provide. Works examining network emulator characteristics include [15] which focus on the delay characteristics, and [10] which considers the throughput characteristics. A comparison of Dummynet, NistNet and NetEm is provided in [12]. An evaluation focused on NetEm is provided in [7].

6 Conclusions and Future Work

Deterministic emulation has been useful for a range of protocol mechanism evaluations and implementation validations. Allowing precise control of the placement of emulation effects allows detailed examination of protocol implementation behavior, and may also have benefits from a statistical perspective. This paper has provided a first overview of the Linux-based KauNetEm deterministic emulation capability, which is built on top of the NetEm emulation infrastructure.

We consider KauNetEm to be an ongoing effort. As part of our future work, we intend to improve the handling of multiple simultaneous deterministic effects, and implement a new trigger pattern effect.

References

[1] Carbone, M., and Rizzo, L. 2010. Dummynet revisited. *ACM SIGCOMM Computer Communication Review* 40(2):12–20.

[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] Garcia, J.; Alfredsson, S.; and Brunstrom, A. 2006. The impact of loss generation on emulation-based protocol evaluation. In *Parallel and Distributed Computing and Networks*, 231–237.

[4] Garcia, J.; Hurtig, P.; and Brunstrom, A. 2008. KauNet: A versatile and flexible emulation system. In *Proceedings of the 5th Swedish National Computer Networking Workshop (SNCNW)*.

[5] Gineste, M., and Garcia, J. 2008. Programmable active emulation of wireless systems - a DVB-RCS example. In *Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks and Workshops, 2008. WiOPT 2008. 6th International Symposium on*, 2–7.

[6] Hemminger, S. 2005. Network emulation with NetEm. In *Proceedings of the 6th Australia's National Linux Conference (LCA2005)*, 18–26.

[7] Jurgelionis, A.; Laulajainen, J.; 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.

[8] Linux advanced routing & traffic control. http://lartc.org.

[9] Lochin, E.; Perennou, T.; and Dairaine, L. 2012. When should i use network emulation? *Annals of telecommunications-Annales des télécommunications* 67(5-6):247–255.

[10] Minhas, T. N.; Fiedler, M.; Shaikh, J.; and Arlos, P. 2011. Evaluation of throughput performance of traffic shapers. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International*, 1596–1600.

[11] Neira-Ayuso, P.; Gasca, R. M.; and Lefevre, L. 2010. Communicating between the kernel and user-space in linux using netlink sockets. *Software: Practice and Experience* 40(9):797–810.

[12] Nussbaum, L., and Richard, O. 2009. A comparative study of network link emulators. In *Proceedings of the 2009 Spring Simulation Multiconference*, 85–92.

[13] Pérennou, T.; Brunstrom, A.; Hall, T.; Garcia, J.; and Hurtig, P. 2011. Emulating opportunistic networks with KauNet triggers. *EURASIP Journal on Wireless Communications and Networking* 2011:4.

[14] Salim, J.; Khosravi, H.; Kleen, A.; and Kuznetsov, A. 2003. Linux netlink as an IP services protocol. RFC 3549, RFC Editor.

[15] Shaikh, J.; Minhas, T. N.; Arlos, P.; and Fiedler, M. 2010. Evaluation of delay performance of traffic shapers. In *Security and Communication Networks (IWSCN), 2010 2nd International Workshop on*, 1–8.