Faculty of Economic Sciences, Communication and IT
Computer Science

Johan Garcia, Per Hurtig and Anna Brunström

KauNet: Design and Usage

Karlstad University Studies
2008:59

Johan Garcia, Per Hurtig och Anna Brunström

KauNet: Design and Usage

Karlstad University Studies
2008:59

Joahn Garcia, Per Hurtig and Anna Brunström. *KauNet: Design and Usage*

Research Report

Karlstad University Studies 2008:59
ISSN 1403-8099
ISBN 978-91-7063-219-8

Distribution:
Faculty of Economic Sciences, Communication and IT
Computer Science
SE-651 88 Karlstad
+46 54 700 10 00

www.kau.se

Printed at: Universitetstryckeriet, Karlstad 2009

KauNet: Design and Usage

Johan Garcia, Per Hurtig, Anna Brunstrom

February 28, 2009

Abstract

KauNet is an emulation system that allows deterministic placement of packet losses and bit-errors as well as more precise control over bandwidth and delay changes. KauNet is an extension to the well-known Dummynet emulator in FreeBSD and allows the use of pattern and scenario files to increase control and repeatability. This report provides a comprehensive description of the usage of KauNet, as well as a technical description of the design and implementation of KauNet.

Contents

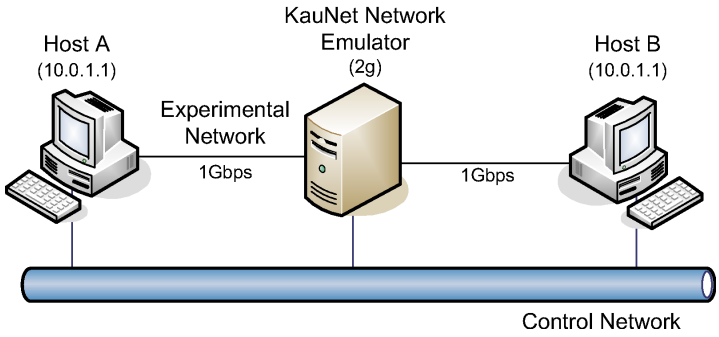


Figure 1: Emulation setup

1 Introduction

Network emulation is commonly used to evaluate and examine the behavior and performance of applications and transport layer protocols. A major advantage of emulation is that a wide variety of real implementations of the applications/protocols to be studied can be used, which is not the case for simulations. An advantage of the KauNet emulation system is that it provides the possibility to perform network emulation with a large degree of control and repeatability. This report describes the design and usage of the KauNet emulation system. The first part of this report focuses on the use of KauNet and the second part covers the design and implementation. A number of appendixes contains various supporting material such as installation instructions, file format specifications and a large number of examples.

It should be noted that this document may be updated from time to time, and a newer version will then be made available on the KauNet site[1]. Furthermore there are, in addition to this document, some documentation and usage examples included in the KauNet source distribution. In case any inconsequential descriptions or examples are found between this document and what is distributed with the source, the material distributed with the source code should be considered more authoritative.

2 Overview and usage example

A typical emulation setup is shown in Figure 1. The KauNet emulator is running on the emulator machine in the middle, emulating the desired network or link characteristics. The protocols/applications that are to be evaluated are installed on the end-node computers Host A and Host B. The emulator machine must use FreeBSD in order to run KauNet since KauNet is an extension to the Dummynet emulation functionality already present in FreeBSD. Host A and Host B can

[1] http://kaunet.sourceforge.net/

however use whatever operating system and application program that is to be tested or evaluated. Preferably, an additional control network is used to for the traffic necessary for experimental setup such as control, configuration, remote starting of applications etc. In order for the traffic to be routed through the network emulator Host A and Host B should be on separate subnets, with routing configured appropriately. In the example shown in Figure 1, Host A is attached to the 10.0.1/24 subnet and Host B is connected to the 10.0.2/24 subnet as part of the experimental network. The interfaces attached to the control network can use whatever IP-addresses, either normal public addresses or private addresses in another subnet. Additional details about this example configuration is provided in Section 2.2.

Before an emulation experiment is performed, the patterns that will control the behavior need to be created and loaded into the kernel of the emulation machine. The creation and handling of patterns are described in greater detail in Section 3.

2.1 KauNet overview

The design of KauNet is centered around a number of pattern-handling extensions to the well known Dummynet emulator, together with user-space programs for pattern creation and management. The use of Dummynet as a starting point provides a stable code-base that has been in wide-spread use for several years, as well as the integration with the `ipfw` program which is used for emulation setup and management. Dummynet has the ability to lose packets, and to apply bandwidth restrictions and delays to the packets, thus emulating the desired link or network conditions. KauNet extends these abilities by also including the ability to introduce bit-errors. Furthermore, KauNet allows deterministic packet losses in addition to the probabilistic losses provided by Dummynet. In fact, KauNet allows bit-errors, packet losses, and delay and bandwidth changes to be exactly and reproducibly controlled on a per-packet or per-millisecond basis with the use of patterns. By employing fine-grained pattern-based control over the emulated behavior, KauNet thus enables emulation-based experiments with a high degree of control and reproducibility of the emulated conditions. The system is flexible with regards to the origin of the patterns, which can be created from collected traces, previous simulations or analytical expressions. When examining the detailed behavior of transport layer protocols, a set of hand-crafted patterns can be used to explore the functioning of a number of transport protocol mechanisms.

KauNet is implemented in the FreeBSD kernel so the patterns need to be inserted into kernel space at the start of the emulation. Thus the fine-grained behavior of the emulation is under the control of the patterns, but higher-level dynamic events can still be incorporated by using command line tools to dynamically switch between multiple patterns. Some potential uses of KauNet include emulation of hand-over scenarios, transport protocol implementation verification, as well as general transport layer and application layer performance evaluations.

2.2 Usage example

This section provides a walk-through example of how KauNet and an example emulation setup can be configured. Of course, this is only one example of a setup and should be viewed only as an illustrative example[2]. The network topology used in this example is the same as shown in Figure 1 earlier, with an experimental network and a control network. The experiment is controlled from Host A (10.0.2.1). For example, the KauNet network emulator (2g) is configured from Host A using ssh over the control network. In this example the emulated characteristics are placed only on the traffic going from Host B (10.0.1.1) to Host A over the experimental network. In order to support the separation of experimental traffic and control traffic, separate networks are configured. Thus the emulator machine needs three network interfaces. The first interface (em0) is connected to the control network. The other interfaces (em1 and em2) are connected directly to Host A and Host B using a crossed Ethernet cable. Below is a snapshot of the FreeBSD configuration file `/etc/rc.conf`, showing the required network configuration of the emulator.

```
gateway_enable="YES"
hostname="2g"
ifconfig_em0="inet <IP-ADDRESS> netmask 255.255.255.0"
ifconfig_em1="inet 10.0.2.2 netmask 255.255.255.0"
ifconfig_em2="inet 10.0.1.2 netmask 255.255.255.0"
```

First of all, the network emulator needs to function as a gateway, routing traffic between Host A and Host B. This is accomplished by the first line in the configuration file. Secondly, the three network interfaces are configured. Interface em0 is configured with its IP-address on the configuration network, which typically is a regular public IP address, and an appropriate netmask. Interfaces em1 and em2 are configured to belong to the different subnets 10.0.1.0/24 and 10.0.2.0/24. Note that the use of the 10.x.x.x address range requires the subnet masks to be set to the correct values (i.e 255.255.255.0 or /24 depending on the notation).

Furthermore, Host A and Host B are also required to route the experimental data towards the emulator. To accomplish this, their network interfaces must belong to one of the previously mentioned subnets (10.0.1/24 and 10.0.2/24). They must also set static routes to each other via the network emulator. How to set up such an arrangement depends on the operating system used in Host A and B. In our example Linux was used, and the syntax for Host A to set up a static route to Host B that is routed via the network emulator is the following:

```
root@hostA:~$ ifconfig eth0 10.0.2.1 netmask 255.255.255.0 up
root@hostA:~$ route add -net 10.0.1.0/24 gw 10.0.2.2 dev em0
```

[2]Before configuring the network, KauNet should be installed according to the instructions in Appendix A.

The first command initializes the network interface eth0 by giving it an IP-address of 10.0.2.1. The second command adds a line in the routing table. This line tells Host A that all traffic destined to the network 10.0.1.0/24 (which Host B belongs to) should be routed towards IP-address 10.0.2.2 (the network emulator). The syntax for setting up a static route from Host B to Host A is similar. Note that the use of private IP-addresses for the experimental network and public for the control network reduces the risk that the control network inadvertently is used to route experimental packets if a mistake is done during route setup.

After the routing has been setup, the emulator can now be configured to make an initial test. First, the `patt_gen` utility is used to generate a packet loss pattern. To generate the packet loss pattern the `-pkt` switch is used together with the `-pos` switch to specify that the positions will be explicitly provided. The name of the generated pattern loss pattern file is `test1.plp`, and it is a data-driven pattern covering 20 packets. The positions for the losses are packet 5,10 and 15. The resulting `patt_gen` command thus becomes:

```
root@2g:~$ ./patt_gen -pkt -pos test1.plp data 20 5,10,15
```

Then the firewall is configured to flushing out any old configurations that may be left and add a default allow rule to allow general traffic:

```
root@2g:~$ ipfw -f flush
root@2g:~$ ipfw -f pipe flush
root@2g:~$ ipfw add allow all from any to any
```

The next step is to create a firewall rule that routes traffic to a pipe where the emulation takes place:

```
root@2g:~$ ipfw add 1 pipe 100 icmp from 10.0.1.1 to 10.0.2.1 in
```

The pipe must now be configured with the emulated conditions. In this example the command uses the `delay` keyword to set a static delay of 10ms, and the `bw` keyword to set a static bandwidth of 1Mbit/s. The packet losses are configured by using the `pattern` keyword to load the packet loss pattern stored in the test1.plp file (which was generated above).

```
root@2g:~$ ipfw pipe 100 config delay 10ms bw 1Mbit/s pattern test1.plp
```

Now it is time to test the emulated conditions. A simple way to examine that the emulator behaves as expected is to use pings. In this case we send pings from Host A to host B. Note that the firewall rule specifies "icmp from 10.0.0.1 ....." so only icmp packets will be sent to the emulation pipe.

```
user@hostA:~$ ping -c 20 10.0.2.1
```

```
PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.2.1:  icmp_seq=1 ttl=63 time=10.3 ms
64 bytes from 10.0.2.1:  icmp_seq=2 ttl=63 time=10.7 ms
64 bytes from 10.0.2.1:  icmp_seq=3 ttl=63 time=10.6 ms
64 bytes from 10.0.2.1:  icmp_seq=4 ttl=63 time=10.5 ms
64 bytes from 10.0.2.1:  icmp_seq=6 ttl=63 time=10.4 ms
64 bytes from 10.0.2.1:  icmp_seq=7 ttl=63 time=10.3 ms
64 bytes from 10.0.2.1:  icmp_seq=8 ttl=63 time=11.2 ms
64 bytes from 10.0.2.1:  icmp_seq=9 ttl=63 time=11.1 ms
64 bytes from 10.0.2.1:  icmp_seq=11 ttl=63 time=11.0 ms
64 bytes from 10.0.2.1:  icmp_seq=12 ttl=63 time=10.9 ms
64 bytes from 10.0.2.1:  icmp_seq=13 ttl=63 time=10.8 ms
64 bytes from 10.0.2.1:  icmp_seq=14 ttl=63 time=10.7 ms
64 bytes from 10.0.2.1:  icmp_seq=16 ttl=63 time=10.6 ms
64 bytes from 10.0.2.1:  icmp_seq=17 ttl=63 time=10.5 ms
64 bytes from 10.0.2.1:  icmp_seq=18 ttl=63 time=10.4 ms
64 bytes from 10.0.2.1:  icmp_seq=19 ttl=63 time=10.3 ms
64 bytes from 10.0.2.1:  icmp_seq=20 ttl=63 time=10.3 ms

--- 10.0.2.1 ping statistics ---
20 packets transmitted, 17 received, 15% packet loss, time 19002ms
rtt min/avg/max/mdev = 10.307/10.669/11.269/0.291 ms
```

This simple example shows the ability to easily place packet losses at specific positions in a packet stream. As can be seen from the pattern creation command above, the pattern will make KauNet lose packets number 5, 10, and 15 in the incoming packet stream. The output from ping also verifies that this is the case. The packets with sequence numbers 5, 10, and 15 are indeed lost[3]. Additional usage examples are shown in Appendix B.

[3]Some ping implementations have sequence numbers starting from 0. In this case the lost sequence numbers will be 4,9 and 14.

3 Emulation patterns and scenarios file

As mentioned before, patterns are used to control the behavior of KauNet. Currently, patterns can be used to control four different characteristics:

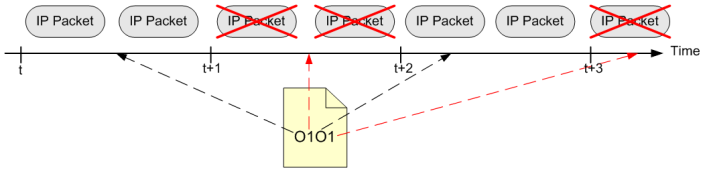
- Packet loss. The packet loss patterns control which packets to lose. This can be done either by specifying the packets that should be lost, or by specifying during which periods in time packet loss should occur.
- Bit-errors. KauNet has the ability to flip individual bits in a data transfer. The bit-error patterns control which bits to change. The patterns can be specified to flip bits at specific positions in the transferred data regardless of when it is sent, or be specified to occur at specific points in time.
- Bandwidth changes. The bandwidth change patters control the evolution of the emulated bandwidth. Changes in bandwidth can be specified to occur when some specific number of packets have been sent, or at specific points in time.
- Delay change patterns. Delay change patterns control the emulated propagation delay. Changes in delay can be specified to occur when some specific number of packets have been sent, or at specific points in time.

As is clear from the description above, patterns can be used in two different modes. When used in the the time-driven mode, KauNet controls the emulated behavior on a per-millisecond[4] basis. The data-driven mode on the other hand controls the behavior on a per-packet basis. An illustration of time-driven versus data-driven packet losses are provided in Figure 2.

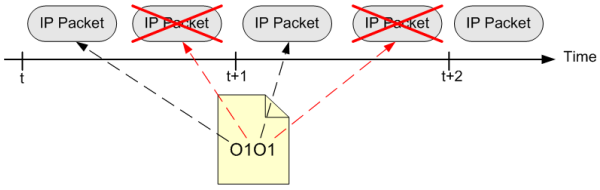
In essence, the mode controls how to move the index forward in the pattern file. The time-driven mode advances the index once per millisecond regardless of whether or not any data is transferred. For time-driven bit-errors the amount of movement by the index is coupled to the bandwidth restriction used in the same pipe. The bandwidth is used to indicate how many bits to process per millisecond. Other kinds of time-driven patterns do not require a bandwidth restriction to be set since they work only on a millisecond level and individual bits do not need to be accounted for. For data-driven patterns the index move forward one step for each packet, except for bit-error patterns where the index is increased according to the number of bits in the packet.

Compressed patterns are created ahead-of-time and then inserted into the kernel space under the control of KauNet. During emulation these patterns are then played to control the emulated behavior. During emulation setup the `ipfw` command can be used to create multiple rules that specify that different flows should be sent to different pipes, which in turn have different patterns. This allows multiple connections between multiple hosts to be emulated using many different patterns.

[4]This is the typical time resolution, using a 1000Hz kernel tick rate. It is also possible to use other resolutions such as $100\mu s$ or 10 ms.



(a) Time-driven packet loss



(b) Data-driven packet loss

Figure 2: Time- versus Data-driven packet loss

It is possible to specify that a pipe should use the same pattern file as used by another pipe for the specified pattern type. Note that if the specified pipe number is itself redirected, the pattern redirected to is used, i.e. redirections are transitive. Each pipe has its own independent pointer into its current position in the shared pattern.

It is also possible to specify a new pattern that should be used once the currently running pattern is exhausted. When the end of the current pattern is reached, the default behavior is to wrap-around and start from the beginning of the same pattern.The pattern files are stored and imported into the kernel in a compressed format. The `patt_gen` utility natively generates this compressed format, but it can also import uncompressed numerical lists or binary files and generate compressed pattern files from these.

As shown in the KauNet system overview in Figure 3, there are several different sources for patterns. Depending on the needs of the user, patterns can be created in the most suitable way. The patterns can potentially be very large. Patterns that have a compressed size of several hundred Megabytes have been successfully tested.

3.1 Pattern file generation

3.1.1 Command line pattern generation tool (patt_gen)

The `patt_gen` command line tool has been developed to create and manage patterns. The tool can generate patterns according to several parametrized distributions. It is also capable of importing uncompressed pattern descriptions from simple text files. These text files can be generated by arbitrarily complex

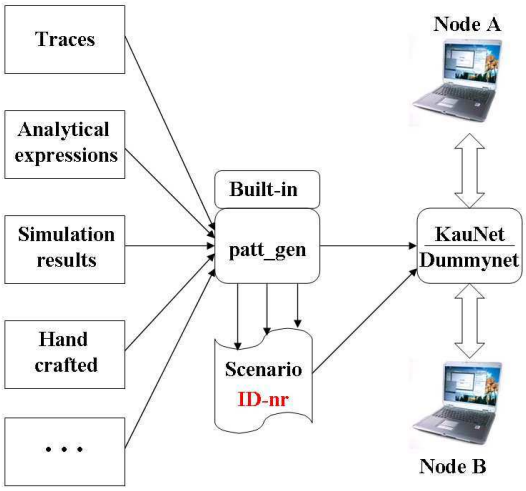


Figure 3: Pattern generation overview

models, off-line simulators or trace collection equipment. The syntax used for the patt_gen toll is illustrated below.

```
patt_gen -ber|-pkt -rand <filename> <mode> <size> <Random_seed> <VAL>

patt_gen -ber|-pkt -ge   <filename> <mode> <size> <Random_seed> <Good_ER>
<Bad_ER> <Good_tran_prob> <Bad_tran_prob>
patt_gen -ber|-pkt -pos  <filename> <mode> <size> <positions>
patt_gen -ber|-pkt -pos  <filename> <mode> <size> -f <infilename>
patt_gen -ber|-pkt -pos  <filename> <mode> <size> -r <infilename>
patt_gen -ber|-pkt -int  <filename> <mode> <size> <interval-list> [<start_value>]
patt_gen -ber|-pkt -int  <filename> <mode> <size> -f <infilename> [<start_value>]
patt_gen -del|-bw -pos  <filename> <mode> <size> <position-values>

patt_gen -del|-bw -pos  <filename> <mode> <size> -f <infilename>
patt_gen -mkscn <scnfile> [-i <id#>] [-t <infotext>] -f <filename>[<filename>]
patt_gen -export <scnfile>
patt_gen -info  <pattfile>|<scenariofile>
patt_gen -dump  <pattfile>|<scenariofile>
patt_gen -graph  [-o <outputfile>] [-p/-l[plots per page]] <pattfile>
[<pattfile>]
patt_gen -graph <pattfile>|<scenariofile>
```

The first switch for the command controls which type of pattern should be produced: bit-error, packet loss, delay change or bandwidth change.
The second switch controls how the pattern should be generated: (pseudo)-

Pattern type	Size unit for data driven mode	Size unit for time driven mode
Bit-error	Kilobyte	Kilobyte
Packet loss	Packets	Milliseconds
Delay change	Packets	Milliseconds
Bandwidth change	Packets	Milliseconds

Table 1: Size unit

randomly, using the Gilbert-Elliot model, or by explicitly giving positions either directly or in a file.
Then follows two parameters which always have the same meaning when generating patterns:

`<filename>` The name of the output file. The suggested suffixes are .bep, .plp, .delp and bwp for bit-error, packet-loss, delay change and bandwidth change patters respectively.

`<mode>` Specifies the generation mode (`data` / `time`). The mode specification can influence the interpretation of later parameters.

The `<size>` parameter specifies the length of the generated pattern. The unit of the size can be either kilobytes, packets or milliseconds dependent on the type of pattern generated as shown in Table 1. This parameter is described separately for each pattern type. Patterns are stored in a compressed format so there is only an indirect connection between the size parameter and the actual disk space needed to store the pattern.

To Pattern information can be obtained by the `-info`, `-dump` and `-graph` switches. The `-info` switch displays various pattern information, and the `-dump` switch additionally dumps the actual pattern info as a list to standard out. If standard out is redirected to a file, this file can then be edited and later imported to patt_gen using the `-f` switch.

Using the `-graph` switch it is possible to generate gnuplot commands that will provide a graphical representation of the pattern when run through gnuplot.

The `-mkscn` and `-export` switches realte to scenario handling and are described in section 3.2.

3.1.2 Pattern generation GUI (pg_gui)

To complement the `patt_gen` tool a GUI called `pg_gui` has been developed. The `pg_gui` program requires tcl/tk and gnuplot to be installed. These are installed by default in many Unix distributions. Note that `patt_gen` and `pg_gui` can be used in most Unix dialects, whereas the KauNet emulation core only runs on FreeBSD. The appearance of `pg_gui` is shown in Figure 4, where the panels for bit-error pattern generation are shown. From Figure 4 it can be seen

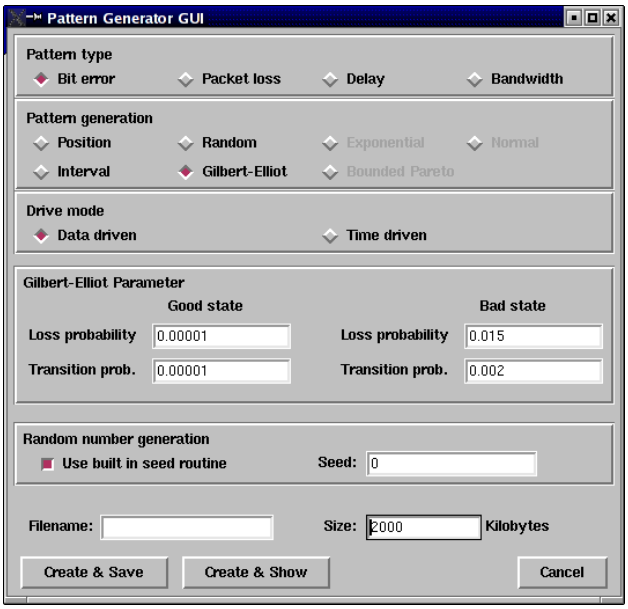


Figure 4: The pg_gui pattern creation GUI

that there are several possible generation functions, but only some of them are active for bit-errors. Using the GUI it is also possible to visually view the generated patterns and interactively explore various pattern generation parameters. Figure 5 shows an example of the pattern generated by the settings shown in Figure 4. The figure shows the distribution of bit-errors in the 2 MB file, and the aggregation of bit-errors caused by the Gilbert-Elliot parameters can be seen in the vertical error streaks. A considerable part of the power and flexibility of the pattern approach, however, lies in its ability to import patterns from other sources via the `patt_gen` command line tool.

3.2 Scenario files

One pattern is necessary for each of the controllable aspects, i.e. bit-errors, packet loss, bandwidth changes and delay changes. Since experiments may require simultaneous control of several aspects, multiple patterns need to be managed. Consider for example the case of a handover, where the worsening link conditions that appear as a node moves away from an access point might increase both delays and losses, and possibly also induce bit-errors depending on the specific emulated technology. To simplify the management of pattern files and to allow simple packaging of several pattern files, a scenario file format has been defined. A scenario file is a concatenation of several pattern files with an additional header. The scenario file header includes a scenario ID (SID) and a free text field that contains a textual description of the scenario. Scenario files

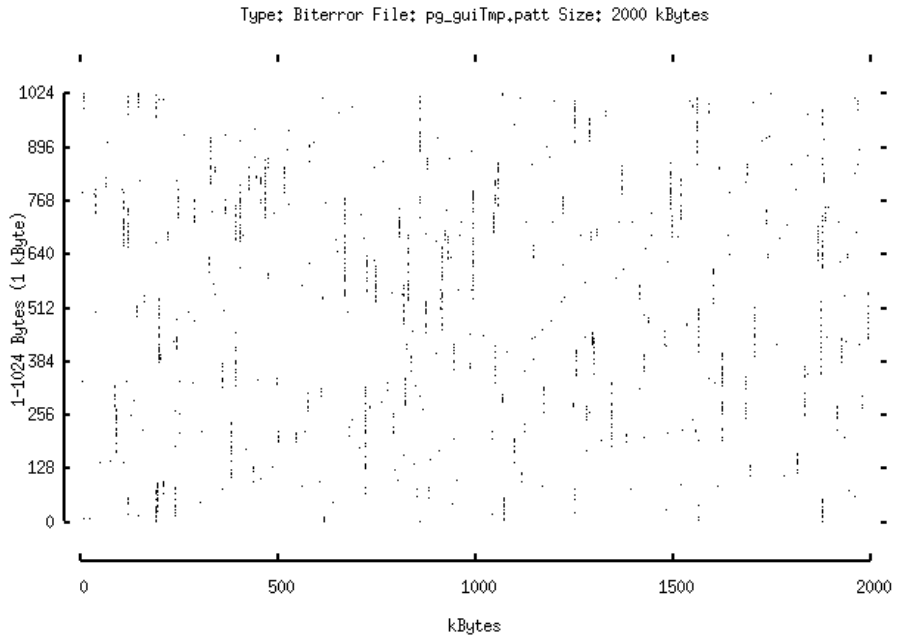


Figure 5: A visualized Gilbert-Elliot bit-error pattern

are also created using the `patt_gen` utility, and the user specifies the pattern files, the SID and the free text. The SIDs are of special interest as the program currently is designed to only accept SIDs which include a correct checksum digit. The idea is that the SIDs should be used by users who wish to make their scenario files publicly available in order to share scenarios or simplify replication of their experiments. We intend to distribute series of globally unique SIDs to any interested researcher, and set up a repository of scenario files and related scenario meta-information where users can share their scenario files.
To generate a scenario file from a collection of pattern files the `-mkscn` switch is defined and to export pattern files from a scenario file the `-export` switch is defined with the following syntax:

```
patt_gen -mkscn <scenariofile> [-i <SID>] [-t "Info text"] -f <patternfile>+
```

`<scenariofile>` is the file that all patterns are collected in. The `-i` switch is optional and must be followed by number. If no `-i` switch is specified a default number will be assigned. The `-t` switch allows for a comment to be inserted, and is also optional. Finally, `-f` must be used to denote that the following arguments are pattern files.

```
patt_gen -export <scenariofile>
```

Simple enough, `-export` extracts the pattern files from the specified scenariofile and writes them to separate pattern files with the same name as the scenario file and a number.

4 KauNet configuration and usage

The configuration of KauNet is similar to the configuration of Dummynet, with some additional keywords added to support the specific functionality of KauNet. When creating an emulation setup there are always two steps that need to be done in order to configure it correctly. Both steps use the inbuilt firewall configuration program `ipfw`, which also configures the emulation control parameters.

The first step is to configure the packet filtering that selects the packets that will be subjected to the emulated conditions. This selection is done using regular firewall rules that can filter out packets on various information such as sender or receiver IP address, port numbers, etc. The packets that match the configured filter settings are then forwarded to a pipe.

The second step is to configure the pipe. The pipe performs delays, drops or corrupts the packets according to its configuration. Both these steps must be performed to get a working emulation system, as was shown in the step-by-step example of Section 2.2.

4.1 Firewall configuration

The configuration of the firewall is done with regular firewall configuration commands. Simple examples of the firewall configuration is provided in the example in Section 2.2 and Appendix B. A more detailed description of the firewall configuration is outside the scope of this paper. It is recommended that the man page of `ipfw` is consulted for additional details. When performing emulation and the emulating machine is acting as a router, it is important to be aware of the correct usage of the `in` and `out` keywords for the firewall rules. If these keywords are not used, it is possible that the packets will be subjected to the the emulated conditions twice, once going in through the network stack on the emulator machine and once going out through the network stack. This behavior is typically not the desired one, so the `in` or `out` keyword should be used to select at witch point the firewall rule should match. This is discussed in more detail in the "packet flow" section of the `ipfw` man page, which also covers how Dummynet/KauNet can be provide emulation when the machine is used not as a router but as a bridge.

4.2 Enhanced pipeconfig syntax

The pipe configuration controls the emulated conditions and which patterns to use. The `ipfw` man page section "traffic shaper (dummynet) configuration" describes how to configure the pipe setup for dummynet. There is also a short man page on dummynet, but that mainly provides a list of associated compilation

options that are relevant when installing dummynet, but do not provide any information on usage.

In addition to the dummynet keywords already defined for pipe configuration KauNet adds the following possible configuration keywords:

`pattern <filename> [timedr|datadr] [nosync]`

`pindex <pipe number> ber|pkt|del|bw [timedr|datadr] [nosync]`

`appendpattern <filename>`

These could be used multiple times on the same command line.

`pattern <filename>`
: This specifies the pattern file that should be used for setting up a pattern. The pattern file header contains information on the type of pattern it is, so there is no need to specify the type explicitly.

`pindex <pipe number> ber|pkt|del|bw`
: This specifies that a pipe should use the same pattern file as used by another pipe for the specified patterntype. Note that if the specified pipe number is itself redirected, the pattern redirected to is used, i.e. redirections are transitive. Example:
```
pipe 100 config pattern patt1.bep
pipe 200 config pattern patt2.bep
pipe 100 config pindex 200 ber
pipe 300 config pindex 100 ber
```
will thus cause pipe 300 to use patt2.bep. The pattern patt1.bep will no longer be in memory since it is not used by any pipe after the `pipe 100 config pindex 200 ber` command. Setting pindex to the same number as the pipe being configured is not allowed (eg `pipe 100 pindex 100 ber`).

`appendpattern <filename>`
: This specifies a pattern that should be used once the currently running pattern is exhausted. When the end of pattern is reached, the default behavior is to wrap-around and start from the beginning of the same pattern. The appendpattern command allows a new pattern to be specified that will be seamlessly switched over to when the end of the current pattern is reached.

`[timedr|datadr]`
: These optional keywords explicitly specifies which mode should be used to forward the index of the pattern file preceding it in the command line. Normally the mode specified in the pattern header is used, and this parameter is left out. In some instances it may be desirable to override the mode specified in the pattern file, which is possible with these keywords. The time-driven mode advances the index as time passes by, regardless of whether or not any data is

transferred. Time-driven bit-errors are coupled to the bandwidth restriction used by the same pipe to determine how many bits to process per millisecond. Other kinds of time-driven patterns do not require a bandwidth restriction to be set since they work only on a millisecond level and individual bits do not need to be accounted for.

`[nosync]` This keyword specifies that forwarding of the index coupled to the time-driven pattern file preceding it in the command line should start immediately, and not wait for the first packet to be transferred. The default behavior is to wait until the first packet is received, and then start the clock that controls the forwarding of time-driven patterns. This keyword allows to override this behavior and start the clock when the pattern is configured.

5 Design

KauNet has been implemented as a kernel extension to the FreeBSD kernel and patches to the `ipfw` system utility. In addition there is also the free-standing programs `patt_gen` (in c) and `pg_gui` (in tcl/tk). A description of how to install KauNet is provided in Appendix A. This section provides some additional details on how KauNet has been implemented. This information can be useful to deepen the understanding of KauNet, and is also relevant when configuring complex emulation setups that have several dynamically changing components.

When creating an emulation setup it is vital to have a good understanding of the system that is to be emulated so that the emulated behavior can match as closely as possible to what is desired. Consider for example the case where both bandwidth restrictions an packet losses occur. Should the packets that will be lost also be considered when the bandwidth restriction is applied, or should they not? Depending on what is emulated either could be correct. If the emulator is set up to emulate network conditions in a (highly multiplexed) network where packet losses occur before a constrained link, then the losses would be best placed before the bandwidth restriction. On the other hand, it KauNet is used to emulate a wireless link that can detect transmission errors and simply drops erroneous frames, the losses would be best placed after a bandwidth restriction.

If emulating a wireless link with link layer retransmissions then the link layer losses will be transformed into bandwidth variations as the link layer retransmissions consumes link resources.

5.1 Implementation of emulated effects

This section describes the overall processing performed by the Dummynet queues, and also gives more detailed information on how the KauNet extensions are implemented. A schematic overview of the Dummynet/KauNet delay components

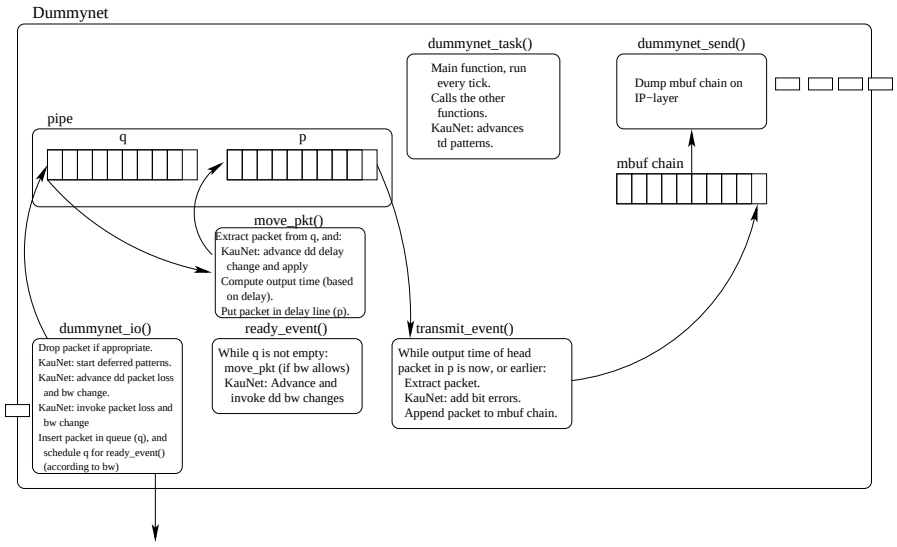


Figure 6: KauNet/Dummynet Queues

and the associated queues are shown in Figure 6 together with some additional text information.

As shown in the figure, a Dummynet/KauNet pipe makes use of two queues:

1. The reception queue, or bandwidth queue, which is called q. In this queue packets coming from the network wait for a duration corresponding to the transmission time for the pipe's emulated bandwidth. If the incoming traffic has a higher rate than the emulated bandwidth, q will start to build up, and the packets will be subjected to a queuing delay in addition to the transmission delay. If q becomes full, it will overflow and packets will be dropped. The queue size for q is configurable in Dummynet, but has a rather conservative maximum of 100 packets.

2. The delayline queue, which is called p, is where packets are put to wait the emulated (propagation) delay that has been configured for the pipe. This queue cannot overflow. In most cases the emulated delay as applied in the delayline does not vary, instead bandwidth variations on short time scales will model delay variations, as for example when the delay effects of link layer retransmissions are emulated. One emulation case where the propagation delay does change is when vertical handovers are emulated.

As KauNet is capable of emulating several effects on the traffic, the relative order of how these effects are applied by KauNet is relevant when several of them

are used in the same pipe The emulated effects are applied in the following order: packet loss, bandwidth restriction (which gives a transmission delay), delay, and finally bit-errors. This can be elaborated:

Packet loss This occurs when a packet is received by the Dummynet layer for a specific pipe. This occurs before any queuing, bandwidth restriction, or delays have been applied to the packet. Losses due to overflow of the emulated queue have, however, occurred.
This emulated effect is realized in the dummynet_io function that is called when a packet is received by the Dummynet subsystem.

Bandwidth restriction This delay is calculated when a packet has finished being in the reception queue (q) behind other packets, and is about to be scheduled with the transmission delay caused by a restricted bandwidth. If a packet should have a packet specific bandwidth value this is where the bandwidth change takes place.
Since the calculation of the transmission delay occurs in two different places depending on whether or not the queue is empty, this action is called from two different places in the code. If q is empty, it is called in the dummynet_io function. If q is not empty, it is called in the ready_event function that is called from dummynet_task when a packet has been forwarded to p and a new packet is scheduled for the transmission delay expiration time. In addition, if the configured bandwidth is so high that several packets are to be sent in the same time-slot, it is called for each packet in that time-slot. If per-packet bandwidth changes should occur when the bandwidth is so high that there are multiple packets per time-slot, care needs to be taken to ensure that the correct behavior indeed occurs.

Delay This delay is calculated when the packet has finished waiting its scheduled transmission delay and is about to be put into the delayline (p) queue (i.e for the emulated propagation delay). At this point the emulated bandwidth restriction has been imposed, but not the propagation delay. This action is also located in the dummynet_task function.

The above description is a brief explanation of how the various effects of KauNet is realized, and it is also textually described in Figure 6. In the text dd and td refers to data-driven and time-driven, respectively. To get a more complete picture the interested user is recommended to study the relevant functions in the source code.

5.2 Sequencing alternatives

The previous section explained the order in which KauNet applies the emulated effects *for a single pipe*. However, if the environment to be emulated requires a different order of the emulated effects from what is provided by the inbuilt KauNet pipe processing, this can be achieved by using multiple pipes. In this

case each pipe only provides one emulated effect, and the firewall rules are constructed such that the emulated effects have the desired order. The simplest way to achieve this is by using the `in` and `out` firewall keywords. For example, it may for some emulated environment be required that the bandwidth restrictions are applied before the packet loss instead of after as is the KauNet behavior. A simple way to achieve this is to configure one pipe with only bandwidth and another with only packet loss. In this case, if (propagation) delay should also be emulated, it can be placed at either pipe. Two firewall rules are created, the one with the `in` keyword points to the bandwidth pipe, and the one with the `out` keyword points to the packet loss pipe.

It could also be possible to use firewall rule chaining to achieve the same effect, but the `in/out` method has been more tested.

A KauNet installation instructions (for FreeBSD 7.0)

This appendix assumes that you have already installed FreeBSD 7.0 on your computer. If you have not installed FreeBSD yet and need help on this matter, please consult the installation instructions in the FreeBSD handbook `http://www.freebsd.org/doc/en/books/handbook/install.html`. Furthermore, the instructions below also assume an i386 architecture. If your system uses a different architecture, simply replace "i386" with the appropriate abbreviation (e.g. "amd64").

To install KauNet you need to have the kernel sources and the sbin sources installed. If one of the directories `/usr/src/sys` or `/usr/src/sbin` is missing, the necessary source files have not been installed. The easiest way to install them is by running `/usr/sbin/sysinstall` as root, choosing Configure, then Distributions, then src, then base, then sys and sbin.

When you have installed FreeBSD and the necessary source files you can install KauNet by conducting the following steps, as root:

1. Copy KauNet into the source tree and extract the KauNet sources from there:
   ```
   cp /kaunet-1.0.0.tar.gz /usr/src
   cd /usr/src
   tar zxvf kaunet-1.0.0.tar.gz
   ```

   Note: this will overwrite the following configuration files:
   a) `/usr/src/sys/conf/files`
   b) `/usr/src/sys/conf/options`

2. Enter the kernel configuration directory and make a copy of the default kernel configuration:
   ```
   cd /usr/src/sys/i386/conf
   cp GENERIC KAUNET
   ```

3. Modify "KAUNET" to include the following lines:
   ```
   options IPFIREWALL
   options IPFIREWALL_DEFAULT_TO_ACCEPT
   options DUMMYNET
   options KAUNET
   options HZ=1000
   ```

4. Modify the line in "KAUNET" that reads:
   `ident GENERIC`
   to read
   `ident KAUNET`

5. Configure the kernel according to the KauNet configuration file:
   `/usr/sbin/config KAUNET`

6. Compile and install the new KauNet kernel:
   ```
   cd ../compile/KAUNET
   make cleandepend && make depend && make && make install
   ```

7. Restart the computer:
   ```
   reboot
   ```

8. Compile and install the KauNet configuration interface:
   ```
   cd /usr/src/sbin/ipfw
   make ; make install
   ```

9. Compile and install the pattern creation utility patt_gen:
   ```
   cd /usr/src/sbin/ipfw
   gcc -Wall -o patt_gen patt_gen.c
   install patt_gen pg_gui.tcl /usr/bin
   ```

10. Usage examples and tests are also available:
    ```
    cd /usr/src/sbin/ipfw/knet-tests
    ```

When all steps have been completed, KauNet is ready to use. In the next appendix, Appendix B, a number of simple usage examples are provided, to get you started.

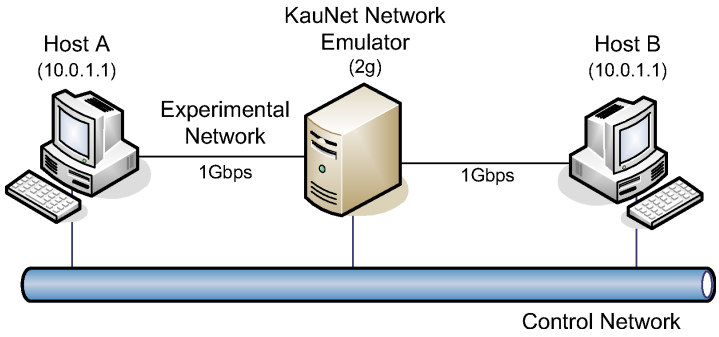


Figure 7: Example Environment

B Additional usage examples

The examples in this section strive to provide an illustration of the use of KauNet, and give some reference to the emulation behavior to be expected for various configurations. The examples show both how to generate different types of emulation patterns and how to configure KauNet to use them. Furthermore, the examples also show some simple experiments using these configurations. However, the examples are by necessity simplified and should only be used to understand how KauNet can be configured, and also to validate the installation. More examples on how to generate specific patterns are available in Appendix C.

The examples are divided into five categories, which follow in the coming five sections of this appendix. The first four categories directly follow the different types of patterns that KauNet can use. That is, bit-errors, packet loss, delay changes, and bandwidth changes. The fifth, and last category, shows an example of how to combine different patterns to create more complex emulation scenarios. Each section starts with an overview of the experiments for that category, to make specific configurations easy to find.

All the examples follow the same basic structure. First, the details of the example are described. That is, which type of emulation effect the example illustrates, which result to except, etc. Finally, the output of the emulation is shown, in two parts. The first part shows all the necessary commands to set up the emulation and execute the experiment. The second part contains the output from these commands. As previously mentioned, all examples show very simplistic scenarios. For instance, the only traffic that is used is traffic generated by `ping`.

Figure 7 depicts the environment that the examples were executed in. All example experiments were controlled from Host A (10.0.2.1). For example, the KauNet-enabled network emulator (2g) was configured from Host A using `ssh` over the control network. Note that in all the coming examples, the emulation effects are placed on the traffic going from Host B (10.0.2.1) to Host A (10.0.1.1).

B.1 Bit-errors

In this first part of the appendix, the focus will be on how to emulate bit-errors. Four examples will be introduced. The first example shows how to create and use data-driven position-based bit-error patterns. The second example extends the first, by showing how to introduce sequences of bit-errors. Example three shows how to introduce time sequences of bit-errors, and the last example is a slight variation of the third.

B.1.1 Position-based bit-errors

This example shows how to insert bit-errors deterministically, in KauNet's data-driven mode. In this example the bit-errors have been placed so that they invert the least significant bit in the id part of two incoming ICMP echo reply headers. The least significant bit of these id fields are bit number 872 and 2216, counting from the first bit that enters the emulator. We also use TCPdump in this example, to show the deatils of the ICMP headers.

```
   user@hostB:~$ ssh root@2g ./patt_gen -ber -pos test1.bep data 1
872,2216
   user@hostB:~$ ssh root@2g ipfw -f flush
   user@hostB:~$ ssh root@2g ipfw -f pipe flush
   user@hostB:~$ ssh root@2g ipfw add allow all from any to any
   user@hostB:~$ ssh root@2g ipfw add 1 pipe 100 icmp from 10.0.1.1
to 10.0.2.1 in
   user@hostB:~$ ssh root@2g ipfw pipe 100 config delay 10ms bw 1Mbit/s
pattern test1.bep
   user@hostB:~$ ssh -f root@2g tcpdump -l -i em1 icmp
   user@hostB:~$ ping -c 4 10.0.1.1

   PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
   64 bytes from 10.0.1.1:  icmp_seq=1 ttl=63 time=11.1 ms
   13:29:33.289842 IP 10.0.2.1 > 10.0.1.1:  ICMP echo request, id 31019,
seq 1, length 64
   13:29:33.300849 IP 10.0.1.1 > 10.0.2.1:  ICMP echo reply, id 31019,
seq 1, length 64
   13:29:34.288865 IP 10.0.2.1 > 10.0.1.1:  ICMP echo request, id 31019,
seq 2, length 64
   13:29:34.299843 IP 10.0.1.1 > 10.0.2.1:  ICMP echo reply, id 31018,
seq 2, length 64
   64 bytes from 10.0.1.1:  icmp_seq=3 ttl=63 time=10.4 ms
   13:29:35.298389 IP 10.0.2.1 > 10.0.1.1:  ICMP echo request, id 31019,
seq 3, length 64
   13:29:35.308837 IP 10.0.1.1 > 10.0.2.1:  ICMP echo reply, id 31019,
seq 3, length 64
```

```
   13:29:36.298546 IP 10.0.2.1 > 10.0.1.1:  ICMP echo request, id 31019,
seq 4, length 64
   13:29:36.308833 IP 10.0.1.1 > 10.0.2.1:  ICMP echo reply, id 31018,
seq 4, length 64

   --- 10.0.1.1 ping statistics ---
   4 packets transmitted, 2 received, 50% packet loss, time 3008ms
   rtt min/avg/max/mdev = 10.498/10.820/11.143/0.338 ms
```

B.1.2 Intervals of bit-errors

This example shows how to insert sequences of bit-errors deterministically, in KauNet's data-driven mode. In this example the bit-errors have been placed so that they invert all the bits of the id fields of two incoming ICMP echo reply headers. The bit ranges of these id fields are [865,881] and [2209,2225], counting from the first bit that enters the emulator.

The output below shows that the range of these bit-errors completely inverts the id fields of the ICMP echo reply messages with sequence numbers 2 and 4.

```
   user@hostB:~$ ssh root@2g ./patt_gen -ber -int test2.bep data 1
865,881,2209,2225 0
   user@hostB:~$ ssh root@2g ipfw -f flush
   user@hostB:~$ ssh root@2g ipfw -f pipe flush
   user@hostB:~$ ssh root@2g ipfw add allow all from any to any
   user@hostB:~$ ssh root@2g ipfw add 1 pipe 100 icmp from 10.0.1.1
to 10.0.2.1 in
   user@hostB:~$ ssh root@2g ipfw pipe 100 config delay 10ms bw 1Mbit/s
pattern test2.bep
   user@hostB:~$ ssh -f root@2g tcpdump -l -i em1 icmp
   user@hostB:~$ ping -c 4 10.0.1.1

   PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
   64 bytes from 10.0.1.1:  icmp_seq=1 ttl=63 time=10.4 ms
   13:29:44.430472 IP 10.0.2.1 > 10.0.1.1:  ICMP echo request, id 36395,
seq 1, length 64
   13:29:44.440797 IP 10.0.1.1 > 10.0.2.1:  ICMP echo reply, id 36395,
seq 1, length 64
   13:29:45.431246 IP 10.0.2.1 > 10.0.1.1:  ICMP echo request, id 36395,
seq 2, length 64
   13:29:45.441790 IP 10.0.1.1 > 10.0.2.1:  ICMP echo reply, id 29140,
seq 2, length 64
   64 bytes from 10.0.1.1:  icmp_seq=3 ttl=63 time=10.6 ms
   13:29:46.431277 IP 10.0.2.1 > 10.0.1.1:  ICMP echo request, id 36395,
seq 3, length 64
   13:29:46.441785 IP 10.0.1.1 > 10.0.2.1:  ICMP echo reply, id 36395,
seq 3, length 64
```

```
   13:29:47.431308 IP 10.0.2.1 > 10.0.1.1:  ICMP echo request, id 36395,
seq 4, length 64
   13:29:47.441780 IP 10.0.1.1 > 10.0.2.1:  ICMP echo reply, id 29140,
seq 4, length 64

   --- 10.0.1.1 ping statistics ---
   4 packets transmitted, 2 received, 50% packet loss, time 3000ms
   rtt min/avg/max/mdev = 10.450/10.539/10.629/0.136 ms
```

B.1.3 Time intervals of bit-errors

In this example we continue to insert bit-errors in sequence. This time, however, the pattern is not data-driven, but instead time-driven. The pattern used in this example is created to invert all bits that enters the emulator 350-600 ms after the first bit has entered. Time-driven bit-error patterns are related to the particular bandwidth that is used. For instance, if a bandwidth of 1 Mbit/s is used, as in this example, then the interval should be set to [350000,650000] to generate a bit-error interval for [350,650] ms. Note the last option passed to the pattern generation program (patt_gen). This is the start value of the pattern. By starting with "0" the bit error period will be turned on at 350 ms and stopped at 650 ms. If "1" had been used as start value instead, the bit error period had started immediately, stopped at 350 ms, and started again at 650 ms.

Note that ping is instructed to transmit an ICMP echo request every 100 ms. As shown in the output, this allows the first four ICMP echo replies to arrive before the bit error period starts. The three following replies are then corrupted, and discarded, due to the bit error period. Finally, after the bit error period, the three last replies are untouched by KauNet.

```
   user@hostB:~$ ssh root@2g ./patt_gen -ber -int test3.bep time 1000
350000,650000 0
   user@hostB:~$ ssh root@2g ipfw -f flush
   user@hostB:~$ ssh root@2g ipfw -f pipe flush
   user@hostB:~$ ssh root@2g ipfw add allow all from any to any
   user@hostB:~$ ssh root@2g ipfw add 1 pipe 100 icmp from 10.0.1.1
to 10.0.2.1 in
   user@hostB:~$ ssh root@2g ipfw pipe 100 config delay 10ms bw 1Mbit/s
pattern test3.bep
   user@hostB:~$ ping -c 10 -i 0.1 10.0.1.1

   PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
   64 bytes from 10.0.1.1:  icmp_seq=1 ttl=63 time=10.7 ms
   64 bytes from 10.0.1.1:  icmp_seq=2 ttl=63 time=11.1 ms
   64 bytes from 10.0.1.1:  icmp_seq=3 ttl=63 time=11.0 ms
   64 bytes from 10.0.1.1:  icmp_seq=4 ttl=63 time=11.1 ms
   64 bytes from 10.0.1.1:  icmp_seq=8 ttl=63 time=11.0 ms
```

```
64 bytes from 10.0.1.1:  icmp_seq=9 ttl=63 time=11.0 ms
64 bytes from 10.0.1.1:  icmp_seq=10 ttl=63 time=11.0 ms

--- 10.0.1.1 ping statistics ---
10 packets transmitted, 7 received, 30% packet loss, time 912ms
rtt min/avg/max/mdev = 10.746/11.030/11.114/0.173 ms
```

B.1.4 Time intervals of bit-errors 2

This example is basically the same as the previous. The only difference is that the ICMP echo requests are sent more frequently, one every 200 ms. The effect of this is that fewer packets are corrupted and thus discarded.

```
user@hostB:~$ ssh root@2g ./patt_gen -ber -int test3.bep time 1000
350000,650000 0
user@hostB:~$ ssh root@2g ipfw -f flush
user@hostB:~$ ssh root@2g ipfw -f pipe flush
user@hostB:~$ ssh root@2g ipfw add allow all from any to any
user@hostB:~$ ssh root@2g ipfw add 1 pipe 100 icmp from 10.0.1.1
to 10.0.2.1 in
user@hostB:~$ ssh root@2g ipfw pipe 100 config delay 10ms bw 1Mbit/s
pattern test3.bep
user@hostB:~$ sudo ping -c 5 -i 0.2 10.0.1.1

PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1:  icmp_seq=1 ttl=63 time=10.6 ms
64 bytes from 10.0.1.1:  icmp_seq=2 ttl=63 time=10.4 ms
64 bytes from 10.0.1.1:  icmp_seq=5 ttl=63 time=10.4 ms

--- 10.0.1.1 ping statistics ---
5 packets transmitted, 3 received, 40% packet loss, time 801ms
rtt min/avg/max/mdev = 10.422/10.510/10.651/0.100 ms
```

B.2 Packet loss

In this part of the appendix, emulation using packet loss patterns will be exemplified. From this category of emulations, seven examples are shown. The first example shows how to create and use data-driven position-based packet loss patterns. The second example extends the first, by showing how to introduce sequences of packet losses. The third example shows an interesting variation of the second example. This example shows how KauNet works when a pattern has been exhausted. The fourth example shows how to generate time sequences of packet losses.

Following these four examples, three additional examples are introduced. These examples show how to perform random-based packet loss with KauNet. The first of these examples shows how to generate uniformly distributed packet losses, which are repeatable between experiments. The second example shows how to create a pattern with a specific number of randomly positioned packet losses. Finally, the third example shows how to create and use packet loss patterns using other random distributions. In this specific example, a gilbert-elliot distribution is used.

B.2.1 Position-based packet loss

This simple example shows the ability to easily place packet losses at specific positions in a packet stream. As can be seen from the pattern creation and configuration, KauNet is instructed to lose packet number 5, 10, and 15 in the incoming packet stream. The output from ping also verifies that this is the case. The packets with sequence numbers 5, 10, and 15 are indeed lost.

```
   user@hostB:~$ ssh root@2g ./patt_gen -pkt -pos test1.plp data 20
5,10,15
   user@hostB:~$ ssh root@2g ipfw -f flush
   user@hostB:~$ ssh root@2g ipfw -f pipe flush
   user@hostB:~$ ssh root@2g ipfw add allow all from any to any
   user@hostB:~$ ssh root@2g ipfw add 1 pipe 100 icmp from 10.0.1.1
to 10.0.2.1 in
   user@hostB:~$ ssh root@2g ipfw pipe 100 config delay 10ms bw 1Mbit/s
pattern test1.plp
   user@hostB:~$ ping -c 20 10.0.1.1

   PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
   64 bytes from 10.0.1.1:  icmp_seq=1 ttl=63 time=10.3 ms
   64 bytes from 10.0.1.1:  icmp_seq=2 ttl=63 time=10.7 ms
   64 bytes from 10.0.1.1:  icmp_seq=3 ttl=63 time=10.6 ms
   64 bytes from 10.0.1.1:  icmp_seq=4 ttl=63 time=10.5 ms
   64 bytes from 10.0.1.1:  icmp_seq=6 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=7 ttl=63 time=10.3 ms
   64 bytes from 10.0.1.1:  icmp_seq=8 ttl=63 time=11.2 ms
```

```
64 bytes from 10.0.1.1:  icmp_seq=9 ttl=63 time=11.1 ms
64 bytes from 10.0.1.1:  icmp_seq=11 ttl=63 time=11.0 ms
64 bytes from 10.0.1.1:  icmp_seq=12 ttl=63 time=10.9 ms
64 bytes from 10.0.1.1:  icmp_seq=13 ttl=63 time=10.8 ms
64 bytes from 10.0.1.1:  icmp_seq=14 ttl=63 time=10.7 ms
64 bytes from 10.0.1.1:  icmp_seq=16 ttl=63 time=10.6 ms
64 bytes from 10.0.1.1:  icmp_seq=17 ttl=63 time=10.5 ms
64 bytes from 10.0.1.1:  icmp_seq=18 ttl=63 time=10.4 ms
64 bytes from 10.0.1.1:  icmp_seq=19 ttl=63 time=10.3 ms
64 bytes from 10.0.1.1:  icmp_seq=20 ttl=63 time=10.3 ms

--- 10.0.1.1 ping statistics ---
20 packets transmitted, 17 received, 15% packet loss, time 19002ms
rtt min/avg/max/mdev = 10.307/10.669/11.269/0.291 ms
```

B.2.2 Intervals of packet loss

Like bit-errors, packet losses can also be specified in intervals. For interval generation each position indicates for which packet the interval state should be changed. The default state is to not lose packets, but in the example below the loss state is immediately entered as the state change is issued for the first packet. Furthermore, we can see that the first packet loss interval is turned off when packet five arrives.

```
user@hostB:~$ ssh root@2g ./patt_gen -pkt -int test2.plp data 20
1,5,12,15
user@hostB:~$ ssh root@2g ipfw -f flush
user@hostB:~$ ssh root@2g ipfw -f pipe flush
user@hostB:~$ ssh root@2g ipfw add allow all from any to any
user@hostB:~$ ssh root@2g ipfw add 1 pipe 100 icmp from 10.0.1.1
to 10.0.2.1 in
user@hostB:~$ ssh root@2g ipfw pipe 100 config delay 10ms bw 1Mbit/s
pattern test2.plp
user@hostB:~$ ping -c 20 10.0.1.1

PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1:  icmp_seq=5 ttl=63 time=10.3 ms
64 bytes from 10.0.1.1:  icmp_seq=6 ttl=63 time=10.2 ms
64 bytes from 10.0.1.1:  icmp_seq=7 ttl=63 time=11.1 ms
64 bytes from 10.0.1.1:  icmp_seq=8 ttl=63 time=11.0 ms
64 bytes from 10.0.1.1:  icmp_seq=9 ttl=63 time=11.0 ms
64 bytes from 10.0.1.1:  icmp_seq=10 ttl=63 time=10.9 ms
64 bytes from 10.0.1.1:  icmp_seq=11 ttl=63 time=10.8 ms
64 bytes from 10.0.1.1:  icmp_seq=15 ttl=63 time=10.5 ms
64 bytes from 10.0.1.1:  icmp_seq=16 ttl=63 time=10.4 ms
64 bytes from 10.0.1.1:  icmp_seq=17 ttl=63 time=10.3 ms
```

```
64 bytes from 10.0.1.1:  icmp_seq=18 ttl=63 time=11.2 ms
64 bytes from 10.0.1.1:  icmp_seq=19 ttl=63 time=11.2 ms
64 bytes from 10.0.1.1:  icmp_seq=20 ttl=63 time=11.1 ms

--- 10.0.1.1 ping statistics ---
20 packets transmitted, 13 received, 35% packet loss, time 19010ms
rtt min/avg/max/mdev = 10.249/10.811/11.284/0.380 ms
```

B.2.3 Intervals of packet loss 2

This example employs the same pattern as the previous example. However, in this example ping is instructed to transmit 40 ICMP echo requests, and not 20 as in the previous example. When the pattern is generated, as shown below, the pattern size is set to 20. Thus, when 20 packets have passed through KauNet, the pattern is exhausted. As shown in the example output below, KauNet simply restarts the pattern after such an event. Thus, it is important to create patterns with appropriate size, to avoid that emulation effects become cyclic. In some situations, however, the cyclic behavior might be beneficial as it helps to reduce pattern sizes.

```
user@hostB:~$ ssh root@2g ./patt_gen -pkt -int test3.plp data 20
1,5,12,15
user@hostB:~$ ssh root@2g ipfw -f flush
user@hostB:~$ ssh root@2g ipfw -f pipe flush
user@hostB:~$ ssh root@2g ipfw add allow all from any to any
user@hostB:~$ ssh root@2g ipfw add 1 pipe 100 icmp from 10.0.1.1
to 10.0.2.1 in
user@hostB:~$ ssh root@2g ipfw pipe 100 config delay 10ms bw 1Mbit/s
pattern test3.plp
user@hostB:~$ ping -c 40 10.0.1.1

64 bytes from 10.0.1.1:  icmp_seq=5 ttl=63 time=11.1 ms
64 bytes from 10.0.1.1:  icmp_seq=6 ttl=63 time=11.1 ms
64 bytes from 10.0.1.1:  icmp_seq=7 ttl=63 time=11.0 ms
64 bytes from 10.0.1.1:  icmp_seq=8 ttl=63 time=10.9 ms
64 bytes from 10.0.1.1:  icmp_seq=9 ttl=63 time=10.8 ms
64 bytes from 10.0.1.1:  icmp_seq=10 ttl=63 time=10.7 ms
64 bytes from 10.0.1.1:  icmp_seq=11 ttl=63 time=10.6 ms
64 bytes from 10.0.1.1:  icmp_seq=15 ttl=63 time=10.3 ms
64 bytes from 10.0.1.1:  icmp_seq=16 ttl=63 time=10.2 ms
64 bytes from 10.0.1.1:  icmp_seq=17 ttl=63 time=11.2 ms
64 bytes from 10.0.1.1:  icmp_seq=18 ttl=63 time=11.1 ms
64 bytes from 10.0.1.1:  icmp_seq=19 ttl=63 time=11.0 ms
64 bytes from 10.0.1.1:  icmp_seq=20 ttl=63 time=10.9 ms
64 bytes from 10.0.1.1:  icmp_seq=25 ttl=63 time=10.5 ms
64 bytes from 10.0.1.1:  icmp_seq=26 ttl=63 time=10.4 ms
```

```
64 bytes from 10.0.1.1:  icmp_seq=27 ttl=63 time=10.4 ms
64 bytes from 10.0.1.1:  icmp_seq=28 ttl=63 time=10.3 ms
64 bytes from 10.0.1.1:  icmp_seq=29 ttl=63 time=11.2 ms
64 bytes from 10.0.1.1:  icmp_seq=30 ttl=63 time=11.1 ms
64 bytes from 10.0.1.1:  icmp_seq=31 ttl=63 time=11.1 ms
64 bytes from 10.0.1.1:  icmp_seq=35 ttl=63 time=10.7 ms
64 bytes from 10.0.1.1:  icmp_seq=36 ttl=63 time=10.6 ms
64 bytes from 10.0.1.1:  icmp_seq=37 ttl=63 time=10.5 ms
64 bytes from 10.0.1.1:  icmp_seq=38 ttl=63 time=10.5 ms
64 bytes from 10.0.1.1:  icmp_seq=39 ttl=63 time=10.4 ms
64 bytes from 10.0.1.1:  icmp_seq=40 ttl=63 time=10.3 ms

--- 10.0.1.1 ping statistics ---
40 packets transmitted, 26 received, 35% packet loss, time 39011ms
rtt min/avg/max/mdev = 10.292/10.777/11.252/0.333 ms
```

B.2.4 Time intervals of packet loss

So far, all examples using packet loss patterns have been data-driven. It is also possible to create time-driven packet loss patterns. In this example KauNet is instructed to lose all incoming packets that arrive between 350 ms and 649 ms after the first packet has entered the emulator.

```
user@hostB:~$ ssh root@2g ./patt_gen -pkt -int test3.plp time 1000
350,650 0
user@hostB:~$ ssh root@2g ipfw -f flush
user@hostB:~$ ssh root@2g ipfw -f pipe flush
user@hostB:~$ ssh root@2g ipfw add allow all from any to any
user@hostB:~$ ssh root@2g ipfw add 1 pipe 100 icmp from 10.0.1.1
to 10.0.2.1 in
user@hostB:~$ ssh root@2g ipfw pipe 100 config delay 10ms bw 1Mbit/s
pattern test3.plp
user@hostB:~$ sudo ping -c 10 -i 0.1 10.0.1.1

PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1:  icmp_seq=1 ttl=63 time=11.1 ms
64 bytes from 10.0.1.1:  icmp_seq=2 ttl=63 time=11.1 ms
64 bytes from 10.0.1.1:  icmp_seq=3 ttl=63 time=11.1 ms
64 bytes from 10.0.1.1:  icmp_seq=4 ttl=63 time=11.1 ms
64 bytes from 10.0.1.1:  icmp_seq=8 ttl=63 time=11.0 ms
64 bytes from 10.0.1.1:  icmp_seq=9 ttl=63 time=11.0 ms
64 bytes from 10.0.1.1:  icmp_seq=10 ttl=63 time=11.0 ms

--- 10.0.1.1 ping statistics ---
10 packets transmitted, 7 received, 30% packet loss, time 901ms
rtt min/avg/max/mdev = 11.041/11.099/11.149/0.089 ms
```

B.2.5 Uniformly distributed packet losses

Packet losses are also useful for illustrating the repeatability of losses generated from statistical distributions. In this case uniformly distributed random losses. As can be seen from the output, the specific seed used (654320) causes three packets to be lost. The packets lost are packet number 1, 5, and 18. These loss positions are repeated for a packet loss probability of 0.1 if the same seed is used. In this specific scenario, a specified packet loss probability of 0.1 led to a slightly higher frequency of losses than expected (three instead of two). If many different patterns had been generated the average number of packet losses would have approached two, assuming a packet loss rate of 0.1 and a pattern size of 20.

```
   user@hostB:~$ ssh root@2g ./patt_gen -pkt -rand test4.plp data 20
654320 0.1
   user@hostB:~$ ssh root@2g ipfw -f flush
   user@hostB:~$ ssh root@2g ipfw -f pipe flush
   user@hostB:~$ ssh root@2g ipfw add allow all from any to any
   user@hostB:~$ ssh root@2g ipfw add 1 pipe 100 icmp from 10.0.1.1
to 10.0.2.1 in
   user@hostB:~$ ssh root@2g ipfw pipe 100 config delay 10ms bw 1Mbit/s
pattern test4.plp
   user@hostB:~$ ping -c 20 10.0.1.1

   PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
   64 bytes from 10.0.1.1:  icmp_seq=1 ttl=63 time=10.3 ms
   64 bytes from 10.0.1.1:  icmp_seq=3 ttl=63 time=10.5 ms
   64 bytes from 10.0.1.1:  icmp_seq=4 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=5 ttl=63 time=10.3 ms
   64 bytes from 10.0.1.1:  icmp_seq=7 ttl=63 time=11.2 ms
   64 bytes from 10.0.1.1:  icmp_seq=8 ttl=63 time=11.1 ms
   64 bytes from 10.0.1.1:  icmp_seq=9 ttl=63 time=11.0 ms
   64 bytes from 10.0.1.1:  icmp_seq=10 ttl=63 time=10.9 ms
   64 bytes from 10.0.1.1:  icmp_seq=11 ttl=63 time=10.8 ms
   64 bytes from 10.0.1.1:  icmp_seq=12 ttl=63 time=10.8 ms
   64 bytes from 10.0.1.1:  icmp_seq=13 ttl=63 time=10.7 ms
   64 bytes from 10.0.1.1:  icmp_seq=14 ttl=63 time=10.6 ms
   64 bytes from 10.0.1.1:  icmp_seq=15 ttl=63 time=10.5 ms
   64 bytes from 10.0.1.1:  icmp_seq=16 ttl=63 time=10.5 ms
   64 bytes from 10.0.1.1:  icmp_seq=17 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=18 ttl=63 time=10.3 ms
   64 bytes from 10.0.1.1:  icmp_seq=20 ttl=63 time=11.1 ms

   --- 10.0.1.1 ping statistics ---
   20 packets transmitted, 17 received, 15% packet loss, time 19002ms
   rtt min/avg/max/mdev = 10.348/10.725/11.238/0.312 ms
```

B.2.6 Specific number of randomly distributed packet losses

When a pattern is created, it is also possible to specify a specific number of randomly placed packet losses. In this case, four packet losses are requested in a 20 packet pattern. For the specified seed, the losses occurs at positions 8, 16, 17, and 19. Again, these loss positions are repeated if the same seed is used when generating the pattern.

```
   user@hostB:~$ ssh root@2g ./patt_gen -pkt -rand test5.plp data 20
8246597 4
   user@hostB:~$ ssh root@2g ipfw -f flush
   user@hostB:~$ ssh root@2g ipfw -f pipe flush
   user@hostB:~$ ssh root@2g ipfw add allow all from any to any
   user@hostB:~$ ssh root@2g ipfw add 1 pipe 100 icmp from 10.0.1.1
to 10.0.2.1 in
   user@hostB:~$ ssh root@2g ipfw pipe 100 config delay 10ms bw 1Mbit/s
pattern test5.plp
   user@hostB:~$ ping -c 20 10.0.1.1

   PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
   64 bytes from 10.0.1.1:  icmp_seq=1 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=2 ttl=63 time=10.6 ms
   64 bytes from 10.0.1.1:  icmp_seq=3 ttl=63 time=10.6 ms
   64 bytes from 10.0.1.1:  icmp_seq=4 ttl=63 time=10.5 ms
   64 bytes from 10.0.1.1:  icmp_seq=5 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=6 ttl=63 time=10.3 ms
   64 bytes from 10.0.1.1:  icmp_seq=7 ttl=63 time=10.2 ms
   64 bytes from 10.0.1.1:  icmp_seq=9 ttl=63 time=11.1 ms
   64 bytes from 10.0.1.1:  icmp_seq=10 ttl=63 time=11.0 ms
   64 bytes from 10.0.1.1:  icmp_seq=11 ttl=63 time=10.9 ms
   64 bytes from 10.0.1.1:  icmp_seq=12 ttl=63 time=10.8 ms
   64 bytes from 10.0.1.1:  icmp_seq=13 ttl=63 time=10.8 ms
   64 bytes from 10.0.1.1:  icmp_seq=14 ttl=63 time=10.7 ms
   64 bytes from 10.0.1.1:  icmp_seq=15 ttl=63 time=10.6 ms
   64 bytes from 10.0.1.1:  icmp_seq=18 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=20 ttl=63 time=11.2 ms

   --- 10.0.1.1 ping statistics ---
   20 packets transmitted, 16 received, 20% packet loss, time 19002ms
   rtt min/avg/max/mdev = 10.263/10.703/11.219/0.290 ms
```

B.2.7 Gilbert-elliot distributed packet losses

Several built-in statistical distributions can be used when creating patterns. In this example a pattern is created using a gilbert-elliot model. With the parameterization and the specific seed used in this example packet losses occur at

positions 17, 18, 19, 22, 24, 25, 26, 37, 48, 71, 75, 76, 78, 81, 82, 84, 85, and 86.

```
   user@hostB:~$ ssh root@2g ./patt_gen -pkt -ge test6.plp data 100
762597 0.05 0.8 0.08 0.15
   user@hostB:~$ ssh root@2g ipfw -f flush
   user@hostB:~$ ssh root@2g ipfw -f pipe flush
   user@hostB:~$ ssh root@2g ipfw add allow all from any to any
   user@hostB:~$ ssh root@2g ipfw add 1 pipe 100 icmp from 10.0.1.1
to 10.0.2.1 in
   user@hostB:~$ ssh root@2g ipfw pipe 100 config delay 10ms bw 1Mbit/s
pattern test6.plp
   user@hostB:~$ sudo ping -c 100 -i 0.1 10.0.1.1

   PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
   64 bytes from 10.0.1.1:  icmp_seq=1 ttl=63 time=10.5 ms
   64 bytes from 10.0.1.1:  icmp_seq=2 ttl=63 time=10.5 ms
   64 bytes from 10.0.1.1:  icmp_seq=3 ttl=63 time=10.5 ms
   64 bytes from 10.0.1.1:  icmp_seq=4 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=5 ttl=63 time=10.5 ms
   64 bytes from 10.0.1.1:  icmp_seq=6 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=7 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=8 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=9 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=10 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=11 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=12 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=13 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=14 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=15 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=16 ttl=63 time=10.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=20 ttl=63 time=10.3 ms
   64 bytes from 10.0.1.1:  icmp_seq=21 ttl=63 time=10.3 ms
   64 bytes from 10.0.1.1:  icmp_seq=23 ttl=63 time=10.3 ms
   64 bytes from 10.0.1.1:  icmp_seq=27 ttl=63 time=10.3 ms
   64 bytes from 10.0.1.1:  icmp_seq=28 ttl=63 time=10.2 ms
   ..........
   <lines deleted due to space restrictions>
   ..........
   64 bytes from 10.0.1.1:  icmp_seq=98 ttl=63 time=10.7 ms
   64 bytes from 10.0.1.1:  icmp_seq=99 ttl=63 time=10.7 ms
   64 bytes from 10.0.1.1:  icmp_seq=100 ttl=63 time=10.7 ms

   --- 10.0.1.1 ping statistics ---
   100 packets transmitted, 82 received, 18% packet loss, time 9913ms
   rtt min/avg/max/mdev = 10.257/10.817/11.278/0.318 ms
```

B.3 Delay change

In addition to "errors" like packet loss and bit-errors, KauNet can also model delay and bandwidth variations. This section exemplifies the emulation of end-to-end delay changes. Three simple delay change scenarios will be shown. The first example shows how to model a decrease in the end-to-end delay, at a specific time. The second example is a variation of the first, using heavier traffic generation and thereby revealing more details on how delay changes are performed in KauNet. Finally, the third example shows how to perform delay increases at certain positions within a traffic flow.

B.3.1 Position-based delay changes

Delay change patterns can also be illustrated using ping. This example illustrates the use of delay changes at two positions. Although the pipe is configured to delay packets for 10 ms the first packet will have a delay of approximately 50 ms. This happens because the first position-value pair in the pattern sets the new value 50 for position 1. When the 10'th packet enters the pipe, the delay is reduced to 10 ms.

```
   user@hostB:~$ ssh root@2g ./patt_gen -del -pos test1.dcp data 20
1,50,10,10
   user@hostB:~$ ssh root@2g ipfw -f flush
   user@hostB:~$ ssh root@2g ipfw -f pipe flush
   user@hostB:~$ ssh root@2g ipfw add allow all from any to any
   user@hostB:~$ ssh root@2g ipfw add 1 pipe 100 icmp from 10.0.1.1
to 10.0.2.1 in
   user@hostB:~$ ssh root@2g ipfw pipe 100 config delay 10ms bw 1Mbit/s
pattern test1.dcp
   user@hostB:~$ sudo ping -c 20 -i 0.1 10.0.1.1

   PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
   64 bytes from 10.0.1.1:  icmp_seq=1 ttl=63 time=50.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=2 ttl=63 time=50.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=3 ttl=63 time=50.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=4 ttl=63 time=50.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=5 ttl=63 time=50.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=6 ttl=63 time=50.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=7 ttl=63 time=50.3 ms
   64 bytes from 10.0.1.1:  icmp_seq=8 ttl=63 time=50.4 ms
   64 bytes from 10.0.1.1:  icmp_seq=9 ttl=63 time=50.3 ms
   64 bytes from 10.0.1.1:  icmp_seq=10 ttl=63 time=10.3 ms
   64 bytes from 10.0.1.1:  icmp_seq=11 ttl=63 time=10.3 ms
   64 bytes from 10.0.1.1:  icmp_seq=12 ttl=63 time=10.3 ms
   64 bytes from 10.0.1.1:  icmp_seq=13 ttl=63 time=10.3 ms
   64 bytes from 10.0.1.1:  icmp_seq=14 ttl=63 time=10.3 ms
```

```
64 bytes from 10.0.1.1:  icmp_seq=15 ttl=63 time=10.3 ms
64 bytes from 10.0.1.1:  icmp_seq=16 ttl=63 time=10.3 ms
64 bytes from 10.0.1.1:  icmp_seq=17 ttl=63 time=10.3 ms
64 bytes from 10.0.1.1:  icmp_seq=18 ttl=63 time=10.3 ms
64 bytes from 10.0.1.1:  icmp_seq=19 ttl=63 time=10.2 ms
64 bytes from 10.0.1.1:  icmp_seq=20 ttl=63 time=10.3 ms

--- 10.0.1.1 ping statistics ---
20 packets transmitted, 20 received, 0% packet loss, time 1901ms
rtt min/avg/max/mdev = 10.288/28.376/50.456/19.938 ms
```

B.3.2 Position-based delay change 2

The second example of delay changes is similar to the previous. However, the packet generation in this example is more intense. In the previous example packets were generated every 100 ms, whereas in this example packets are generated every 10 ms. Thus, when the delay changes, there will be packets left in the delay-line. This example shows the behavior for such scenarios, which is to keep the delay for the packets already being delayed when a decrease occurs, and queue packets with lower delay behind. This semantic ensure that no packet reordering occurs due to delay changes.

```
user@hostB:~$ ssh root@2g ./patt_gen -del -pos test1.dcp data 20
1,50,10,10
user@hostB:~$ ssh root@2g ipfw -f flush
user@hostB:~$ ssh root@2g ipfw -f pipe flush
user@hostB:~$ ssh root@2g ipfw add allow all from any to any
user@hostB:~$ ssh root@2g ipfw add 1 pipe 100 icmp from 10.0.1.1
to 10.0.2.1 in
user@hostB:~$ ssh root@2g ipfw pipe 100 config delay 10ms bw 1Mbit/s
pattern test1.dcp
user@hostB:~$ sudo ping -c 20 -i 0.01 10.0.1.1

PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1:  icmp_seq=1 ttl=63 time=51.1 ms
64 bytes from 10.0.1.1:  icmp_seq=2 ttl=63 time=51.1 ms
64 bytes from 10.0.1.1:  icmp_seq=3 ttl=63 time=51.1 ms
64 bytes from 10.0.1.1:  icmp_seq=4 ttl=63 time=51.1 ms
64 bytes from 10.0.1.1:  icmp_seq=5 ttl=63 time=51.1 ms
64 bytes from 10.0.1.1:  icmp_seq=6 ttl=63 time=51.0 ms
64 bytes from 10.0.1.1:  icmp_seq=7 ttl=63 time=51.0 ms
64 bytes from 10.0.1.1:  icmp_seq=8 ttl=63 time=50.9 ms
64 bytes from 10.0.1.1:  icmp_seq=9 ttl=63 time=51.0 ms
64 bytes from 10.0.1.1:  icmp_seq=10 ttl=63 time=35.9 ms
64 bytes from 10.0.1.1:  icmp_seq=11 ttl=63 time=26.1 ms
64 bytes from 10.0.1.1:  icmp_seq=12 ttl=63 time=14.1 ms
```

```
64 bytes from 10.0.1.1:  icmp_seq=13 ttl=63 time=11.1 ms
64 bytes from 10.0.1.1:  icmp_seq=14 ttl=63 time=10.9 ms
64 bytes from 10.0.1.1:  icmp_seq=15 ttl=63 time=11.0 ms
64 bytes from 10.0.1.1:  icmp_seq=16 ttl=63 time=11.1 ms
64 bytes from 10.0.1.1:  icmp_seq=17 ttl=63 time=11.0 ms
64 bytes from 10.0.1.1:  icmp_seq=18 ttl=63 time=10.9 ms
64 bytes from 10.0.1.1:  icmp_seq=19 ttl=63 time=11.1 ms
64 bytes from 10.0.1.1:  icmp_seq=20 ttl=63 time=10.9 ms

--- 10.0.1.1 ping statistics ---
20 packets transmitted, 20 received, 0% packet loss, time 220ms
rtt min/avg/max/mdev = 10.968/31.206/51.145/18.900 ms, pipe 5
```

B.3.3 Position-based delay change 3

This example shows a scenario quite opposite from the previous. Instead of lowering the delay, the delay is increased from 50 ms to 100 ms for the 10th packet. What happens is simply that the delay for this and the following packets increase.

```
user@hostB:~$ ssh root@2g ./patt_gen -del -pos test1.dcp data 20
1,50,10,100
user@hostB:~$ ssh root@2g ipfw -f flush
user@hostB:~$ ssh root@2g ipfw -f pipe flush
user@hostB:~$ ssh root@2g ipfw add allow all from any to any
user@hostB:~$ ssh root@2g ipfw add 1 pipe 100 icmp from 10.0.1.1
to 10.0.2.1 in
user@hostB:~$ ssh root@2g ipfw pipe 100 config delay 10ms bw 1Mbit/s
pattern test1.dcp
user@hostB:~$ sudo ping -c 20 -i 0.1 10.0.1.1

PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1:  icmp_seq=1 ttl=63 time=50.7 ms
64 bytes from 10.0.1.1:  icmp_seq=2 ttl=63 time=50.9 ms
64 bytes from 10.0.1.1:  icmp_seq=3 ttl=63 time=50.9 ms
64 bytes from 10.0.1.1:  icmp_seq=4 ttl=63 time=50.9 ms
64 bytes from 10.0.1.1:  icmp_seq=5 ttl=63 time=50.9 ms
64 bytes from 10.0.1.1:  icmp_seq=6 ttl=63 time=50.8 ms
64 bytes from 10.0.1.1:  icmp_seq=7 ttl=63 time=50.9 ms
64 bytes from 10.0.1.1:  icmp_seq=8 ttl=63 time=50.8 ms
64 bytes from 10.0.1.1:  icmp_seq=9 ttl=63 time=50.8 ms
64 bytes from 10.0.1.1:  icmp_seq=10 ttl=63 time=100 ms
64 bytes from 10.0.1.1:  icmp_seq=11 ttl=63 time=100 ms
64 bytes from 10.0.1.1:  icmp_seq=12 ttl=63 time=100 ms
64 bytes from 10.0.1.1:  icmp_seq=13 ttl=63 time=100 ms
64 bytes from 10.0.1.1:  icmp_seq=14 ttl=63 time=100 ms
```

```
64 bytes from 10.0.1.1:  icmp_seq=15 ttl=63 time=100 ms
64 bytes from 10.0.1.1:  icmp_seq=16 ttl=63 time=100 ms
64 bytes from 10.0.1.1:  icmp_seq=17 ttl=63 time=100 ms
64 bytes from 10.0.1.1:  icmp_seq=18 ttl=63 time=100 ms
64 bytes from 10.0.1.1:  icmp_seq=19 ttl=63 time=100 ms
64 bytes from 10.0.1.1:  icmp_seq=20 ttl=63 time=100 ms

--- 10.0.1.1 ping statistics ---
20 packets transmitted, 20 received, 0% packet loss, time 1900ms
rtt min/avg/max/mdev = 50.788/78.366/100.877/24.847 ms, pipe 2
```

B.4 Bandwidth change

This section examplifies the emulation of bandwidth changes. Two simple scenarios will be shown. The first example shows how to model an increase in the bandwidth, at a specific time. The second example is a variation of the first, using slightly smaller packets.

B.4.1 Position-based bandwidth change

Bandwidth changes can also be illustrated with ping. However, to generate enough traffic to make bandwidth restrictions clearly visible it is necessary to increase the size of the ICMP packets. The initial bandwidth of 1 Mbps creates an additional delay of 12 ms (i.e. in addition to the fixed delay of 10 ms) for each packet. When the tenth packet arrives the bandwidth is reduced to 100 kbps, and the packets are delayed further. As the generation of packets are quite fast the queue will start to build up, which results in a continuous delay increase.

```
   user@hostB:~$ ssh root@2g ./patt_gen -bw -pos test1.bcp data 20
1,1000,10,100
   user@hostB:~$ ssh root@2g ipfw -f flush
   user@hostB:~$ ssh root@2g ipfw -f pipe flush
   user@hostB:~$ ssh root@2g ipfw add allow all from any to any
   user@hostB:~$ ssh root@2g ipfw add 1 pipe 100 icmp from 10.0.1.1
to 10.0.2.1 in
   user@hostB:~$ ssh root@2g ipfw pipe 100 config delay 10ms bw 10Mbit/s
pattern test1.bcp
   user@hostB:~$ sudo ping -s 1460 -c 20 -i 0.1 10.0.1.1

   PING 10.0.1.1 (10.0.1.1) 1460(1488) bytes of data.
   1468 bytes from 10.0.1.1:  icmp_seq=1 ttl=63 time=22.1 ms
   1468 bytes from 10.0.1.1:  icmp_seq=2 ttl=63 time=22.2 ms
   1468 bytes from 10.0.1.1:  icmp_seq=3 ttl=63 time=22.2 ms
   1468 bytes from 10.0.1.1:  icmp_seq=4 ttl=63 time=22.2 ms
   1468 bytes from 10.0.1.1:  icmp_seq=5 ttl=63 time=22.2 ms
   1468 bytes from 10.0.1.1:  icmp_seq=6 ttl=63 time=22.2 ms
   1468 bytes from 10.0.1.1:  icmp_seq=7 ttl=63 time=22.1 ms
   1468 bytes from 10.0.1.1:  icmp_seq=8 ttl=63 time=22.2 ms
   1468 bytes from 10.0.1.1:  icmp_seq=9 ttl=63 time=22.1 ms
   1468 bytes from 10.0.1.1:  icmp_seq=10 ttl=63 time=130 ms
   1468 bytes from 10.0.1.1:  icmp_seq=11 ttl=63 time=149 ms
   1468 bytes from 10.0.1.1:  icmp_seq=12 ttl=63 time=168 ms
   1468 bytes from 10.0.1.1:  icmp_seq=13 ttl=63 time=187 ms
   1468 bytes from 10.0.1.1:  icmp_seq=14 ttl=63 time=206 ms
   1468 bytes from 10.0.1.1:  icmp_seq=15 ttl=63 time=217 ms
   1468 bytes from 10.0.1.1:  icmp_seq=16 ttl=63 time=228 ms
   1468 bytes from 10.0.1.1:  icmp_seq=17 ttl=63 time=247 ms
```

```
1468 bytes from 10.0.1.1:  icmp_seq=18 ttl=63 time=266 ms
1468 bytes from 10.0.1.1:  icmp_seq=19 ttl=63 time=285 ms
1468 bytes from 10.0.1.1:  icmp_seq=20 ttl=63 time=304 ms

--- 10.0.1.1 ping statistics ---
20 packets transmitted, 20 received, 0% packet loss, time 1917ms
rtt min/avg/max/mdev = 22.143/129.424/304.110/104.711 ms, pipe 3
```

B.4.2 Position-based bandwidth change 2

This example is almost identical to the previous. In this example, however, the size of the ICMP packets are slightly smaller. As the packets are smaller, each packet is subjected to a smaller transmission delay, which in turn results in slightly smaller delays for the packets.

```
user@hostB:~$ ssh root@2g ./patt_gen -bw -pos test1.bcp data 20
1,1000,10,100
user@hostB:~$ ssh root@2g ipfw -f flush
user@hostB:~$ ssh root@2g ipfw -f pipe flush
user@hostB:~$ ssh root@2g ipfw add allow all from any to any
user@hostB:~$ ssh root@2g ipfw add 1 pipe 100 icmp from 10.0.1.1
to 10.0.2.1 in
user@hostB:~$ ssh root@2g ipfw pipe 100 config delay 10ms bw 10Mbit/s
pattern test1.bcp
user@hostB:~$ sudo ping -s 1360 -c 20 -i 0.1 10.0.1.1

PING 10.0.1.1 (10.0.1.1) 1360(1388) bytes of data.
1368 bytes from 10.0.1.1:  icmp_seq=1 ttl=63 time=21.7 ms
1368 bytes from 10.0.1.1:  icmp_seq=2 ttl=63 time=21.8 ms
1368 bytes from 10.0.1.1:  icmp_seq=3 ttl=63 time=21.9 ms
1368 bytes from 10.0.1.1:  icmp_seq=4 ttl=63 time=21.8 ms
1368 bytes from 10.0.1.1:  icmp_seq=5 ttl=63 time=21.8 ms
1368 bytes from 10.0.1.1:  icmp_seq=6 ttl=63 time=21.8 ms
1368 bytes from 10.0.1.1:  icmp_seq=7 ttl=63 time=21.8 ms
1368 bytes from 10.0.1.1:  icmp_seq=8 ttl=63 time=21.8 ms
1368 bytes from 10.0.1.1:  icmp_seq=9 ttl=63 time=21.8 ms
1368 bytes from 10.0.1.1:  icmp_seq=10 ttl=63 time=121 ms
1368 bytes from 10.0.1.1:  icmp_seq=11 ttl=63 time=132 ms
1368 bytes from 10.0.1.1:  icmp_seq=12 ttl=63 time=143 ms
1368 bytes from 10.0.1.1:  icmp_seq=13 ttl=63 time=154 ms
1368 bytes from 10.0.1.1:  icmp_seq=14 ttl=63 time=165 ms
1368 bytes from 10.0.1.1:  icmp_seq=15 ttl=63 time=176 ms
1368 bytes from 10.0.1.1:  icmp_seq=16 ttl=63 time=187 ms
1368 bytes from 10.0.1.1:  icmp_seq=17 ttl=63 time=198 ms
1368 bytes from 10.0.1.1:  icmp_seq=18 ttl=63 time=209 ms
1368 bytes from 10.0.1.1:  icmp_seq=19 ttl=63 time=212 ms
```

```
1368 bytes from 10.0.1.1:  icmp_seq=20 ttl=63 time=223 ms

--- 10.0.1.1 ping statistics ---
20 packets transmitted, 20 received, 0% packet loss, time 1909ms
rtt min/avg/max/mdev = 21.768/106.273/223.775/80.134 ms, pipe 3
```

B.5 Composite example

It is also possible to use multiple patterns in a single emulation. This example shows the interaction effects that can occur when a bit error pattern and a packet loss pattern are used at the same time.

The bit error pattern in this example causes the least significant bit in the second and fourth ICMP echo reply message to be flipped. However, as the packet loss pattern causes the fourth packet to be lost, the bit error in the fourth packet is undetectable.

```
   user@hostB:~$ ssh root@2g ./patt_gen -ber -pos testC.bep data 1
872,2216
   user@hostB:~$ ssh root@2g ./patt_gen -pkt -pos testC.plp data 10
4
   user@hostB:~$ ssh root@2g ipfw -f flush
   user@hostB:~$ ssh root@2g ipfw -f pipe flush
   user@hostB:~$ ssh root@2g ipfw add allow all from any to any
   user@hostB:~$ ssh root@2g ipfw add 1 pipe 100 icmp from 10.0.1.1
to 10.0.2.1 in
   user@hostB:~$ ssh root@2g ipfw pipe 100 config delay 10ms bw 1Mbit/s
pattern testC.bep pattern testC.plp
   user@hostB:~$ ping -c 4 10.0.1.1

   PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
   64 bytes from 10.0.1.1:  icmp_seq=1 ttl=63 time=10.3 ms
   19:23:53.211956 IP 10.0.2.1 > 10.0.1.1:  ICMP echo request, id 56380,
seq 1, length 64
   19:23:53.222156 IP 10.0.1.1 > 10.0.2.1:  ICMP echo reply, id 56380,
seq 1, length 64
   19:23:54.213232 IP 10.0.2.1 > 10.0.1.1:  ICMP echo request, id 56380,
seq 2, length 64
   19:23:54.224150 IP 10.0.1.1 > 10.0.2.1:  ICMP echo reply, id 56381,
seq 2, length 64
   64 bytes from 10.0.1.1:  icmp_seq=3 ttl=63 time=10.9 ms
   19:23:55.213261 IP 10.0.2.1 > 10.0.1.1:  ICMP echo request, id 56380,
seq 3, length 64
   19:23:55.224144 IP 10.0.1.1 > 10.0.2.1:  ICMP echo reply, id 56380,
seq 3, length 64
   19:23:56.213291 IP 10.0.2.1 > 10.0.1.1:  ICMP echo request, id 56380,
seq 4, length 64

   --- 10.0.1.1 ping statistics ---
   4 packets transmitted, 2 received, 50% packet loss, time 3001ms
   rtt min/avg/max/mdev = 10.310/10.649/10.988/0.339 ms
```

C Additional pattern generation examples

In this appendix, the syntax for creating patterns are provided. Furthermore, the appendix also contains a number of examples regarding this matter. In the first section the syntax of generating bit-error patterns is covered. In the following section packet loss patterns are covered. The third section shows how to generate delay change patterns, and the last section details generation of bandwidth change patterns.

C.1 Generating bit-error patterns

There are four different ways of generating a bit-error pattern. They are discussed in separate subsections.

C.1.1 Random bit-error patterns

Random bit error patterns have a uniform loss distribution and can be used to make simple evaluations of protocol performance in the presence of bit-errors.

Syntax:

`patt_gen -ber -rand <filename> <mode> <size> <Random_seed> <VAL>`

`<filename>` The name of the output file. The recommended suffix for bit-error pattern files such as generated here is .bep.

`<mode>` Specifies the generation mode (data/time). The mode specification can influence the interpretation of later parameters.

`<size>` Specifies the length of the generated pattern in kilobytes. A value of 10240 will generate a pattern covering 10 megabytes. If this example pattern is applied to packets of 1500 bytes it would cover approximately 7000 packets when used with the data-driven mode. When used in time-driven mode the pattern covers an amount of time that is related to the bandwidth used. For example, if a value of 10240 is used for a link with a bandwidth of 1000 kbps the pattern will cover 80 seconds.

`<Random seed>` Normally set to `0` which causes the program to create a new seed at each invocation using a modulus-based computation involving measurements of network latencies. Network connectivity for ICMP to the Internet is required. Alternatively, this parameter can be used to set a specific seed value to (re-)create a specific random loss sequence.

`<VAL>` This parameter has two possible meanings depending on its value. If VAL is given as a decimal value < 1 then it is used as a bit-error rate (BER). Each bit will have a probability equal to the BER to become inverted. Since the loss generation is probabilistic, the exact amount of bit-errors in a pattern cannot be exactly controlled. If exact control over the

number of bit-errors is desired, a VAL >=1 can be specified. The program then interprets this as the number of bit-errors that should be present in the pattern, and generates a pattern file with exactly this number of randomly placed bit-errors.

Example:

```
[user@testpc1 prgms]$ patt_gen -ber -rand testrandpat1.bep 2048 0 .000008
```

This example generates a bit-error pattern where the losses are uniformly distributed in the pattern file. In this example an internally generated seed is used and the pattern is stored in the file `testrandpat1.bep` which covers 2 Mbytes of traffic. In this particular run 134 bit-errors were generated.

C.1.2 Gilbert-Elliot bit-error patterns

The `patt_gen` program can generate pattern files according to the commonly used Gilbert-Elliot model. This model has two states, a good state and a bad state. Each of the states have a BER attached to it, and a transition probability to enter the other state. This model captures the fading characteristics present in most wireless systems better than the random uniform model.

Syntax:

```
patt_gen -ber -ge   <filename> <mode> <size> <Random_seed> <Good_BER>
<Bad_BER> <Good_tran_prob> <Bad_tran_prob>
```

`<filename>` The name of the output file. The recommended suffix for bit-error pattern files such as generated here is .bep.

`<mode>` Specifies the generation mode (data/time). The mode specification can influence the interpretation of later parameters.

`<size>` Specifies the length of the generated pattern in kilobytes. A value of 10240 will generate a pattern covering 10 megabytes. If this example pattern is applied to packets of 1500 bytes it would cover approximately 7000 packets when used with the data-driven mode. When used in time-driven mode the pattern covers an amount of time that is related to the bandwidth used. Since the packets are stored in a compressed format the actual disk space needed will be much less, exactly how much is dependent on the BER used.

`<Random seed>` Normally set to `0` which causes the program to create a new seed at each invocation using a modulus-based computation involving measurements of network latencies. Network connectivity for ICMP to the Internet is required. Alternatively, this parameter can be used to set a specific seed value to (re-)create a specific random loss sequence.

<Good_BER> This decimal sets the BER that applies when being in the good state.

<Bad_BER> This decimal sets the BER that applies when being in the bad state.

<Good_tran_prob> This decimal sets the probability of transitioning from the good state to the bad state.

<Bad_tran_prob> This decimal sets the probability of transitioning from the bad state to the good state.

Example:

```
[user@testpc1 prgms]$ patt_gen -ber -ge testrandpat2.bep time 2048
0 .000000001 .008 .000002 .0008
```

This example generates a bit-error pattern that groups the bit-errors in a bursty fashion. The length and interval of bursts are controlled by the transition probabilities. In this example an internally generated seed is used and the pattern is stored in the file `testrandpat2.bep` which covers 2 Mbytes of traffic. In this particular run 425 bit-errors were generated. Note that although the resulting BER is higher in this example compared to the example with uniform random losses above, the Packet Error Rate for traffic consisting of 1500 byte TCP packets will actually be lower. When the errors appear in bursts they affect fewer TCP packets than if they are randomly spread.

C.1.3 Fixed position bit-error patterns

With fixed position bit-errors it is possible to place individual bit-errors precisely. This approach is most useful in the data-driven mode where bit errors can be placed at specific positions in headers or the payload to examine for example protocol robustness or payload resilience. In the time-driven mode the placement is done at bit-transfer-times, and hence affected by variability of sending times.

Syntax:

```
patt_gen -ber -pos  <filename> <mode> <size> <positions>
patt_gen -ber -pos  <filename> <mode> <size> -f <infilename>
patt_gen -ber -pos  <filename> <mode> <size> -r <infilename>
```

<filename> The name of the output file. The recommended suffix for bit-error pattern files such as generated here is .bep.

<mode> Specifies the generation mode (data/time). The mode specification can influence the interpretation of later parameters.

`<size>` Specifies the length of the generated pattern in kilobytes. A value of 10240 will generate a pattern covering 10 megabytes. If this example pattern is applied to packets of 1500 bytes it would cover approximately 7000 packets when used with the data-driven mode. When used in time-driven mode the pattern covers an amount of time that is related to the bandwidth used. Since the patterns are stored in a compressed format the actual disk space needed will be much less, exactly how much is dependent on the BER used.

`<positions>` This is a comma separated list which should list the positions of the bit errors in ascending order. The first bit transferred is numbered 1.

`-f <infilename>` Gives the filename of a text-file that contains bit-error positions. The file should contain an ascending list of error positions with one value per line without any whitespace in front of the value and no blank lines.

`-r <infilename>` Gives the filename of a binary file that contains bit-error positions. Each bit in the file corresponds to one bit of data (data-driven) or one bit-time of transmission (time-driven). A bit set to one generates a bit-error at the corresponding position.

Example:

```
[user@testpc1 prgms]$ patt_gen -ber -pos testpospat.bep data 1 872,2216
```

This example generates a bit-error pattern that contains two bit-errors. In this example the pattern is stored in the file `testpospat.bep` which covers only 1 kbyte of traffic. In this particular run bits 872 and 2216 will be corrupted.

C.1.4 Fixed interval bit-error patterns

With fixed interval bit-errors it is possible to place intervals of bit-errors precisely. In the data-driven mode the interval list specifies at which bit each interval starts and ends. The operation is similar for the time-driven mode, except that time units, and not specific bits, are used.

Syntax:

```
patt_gen -ber -int  <filename> <mode> <size> <interval-list> <start_value>
patt_gen -ber -int  <filename> <mode> <size> -f <infilename> <start_value>
```

`<filename>` The name of the output file. The recommended suffix for bit-error pattern files such as generated here is .bep.

`<mode>` Specifies the generation mode (data/time). The mode specification can influence the interpretation of later parameters.

`<size>` Specifies the length of the generated pattern in kilobytes. A value of 10240 will generate a pattern covering 10 megabyte. If this example pattern is applied to packets of 1500 bytes it would cover approximately 7000 packets when used with the data-driven mode. When used in time-driven mode the pattern covers an amount of time that is related to the bandwidth used. Since the packets are stored in a compressed format the actual disk space needed will be much less, exactly how much is dependent on the BER used.

`<interval-list>` Comma separated list which lists the start and endpoints of the intervals.

`-f <infilename>` Gives the filename of a text-file that contains bit-error intervals. The file should contain an ascending list of error interval positions with one value per line without any whitespace in front of the value and no blank lines.

`<start_value>` This value decides if the first initial interval should start without errors (`0`) or with errors (`1`). The default is 0.

Example:

```
[user@testpc1 prgms]$ patt_gen -ber -int testintpat.bep data 1 100,500
0
```

This example generates a bit-error pattern that contains a bit-error period between bit 100 and 500. The pattern is stored in the file `testintpat.bep` which covers only 1 kbyte of traffic. Notice that the start value in this example is 0. If 1 had been used instead, the interval between bit 100 and 500 would have been free from errors. However, the rest of the 1 kbytes of traffic had been subjected to errors.

C.2 Generating packet loss patterns

The syntax for generation of packet loss patterns is similar to the one used for bit-errors, but some differences are necessary due to the change of atomic unit from bit to packet.

C.2.1 Random packet loss patterns

Random packet loss patterns have a uniform loss distribution and can be used to make evaluations of protocol performance in the presence of packet losses. The possibility to have uniform random losses are present in several network emulators such as dummynet and NISTnet. The use of patterns with randomly placed losses in some sense mimics the standard behaviour of the random losses in dummynet, with the important difference that pattern-based random losses allows for later exact repetition of a loss sequence. Random packet loss patterns are typically used with the data-driven mode, where a one inside the pattern corresponds to a loss at that position. For time driven losses a one inside the pattern corresponds to losing all packets sent during a specific millisecond in time. The rest of this subsection will focus on the data-driven behavior.

Syntax:

`patt_gen -pkt -rand <filename> <mode> <size> <Random_seed> <PVAL>`

`<filename>` The name of the output file. The recommended suffix for packet loss pattern files such as generated here is .plp.

`<mode>` Specifies the generation mode (data/time). The mode specification can influence the interpretation of later parameters.

`<size>` The meaning of this parameter is different between the modes. For data-driven it specifies the length of the generated pattern in number of packets. A value of 10000 will generate a pattern covering 10000 packets. For time-driven mode, the size specifies the length of the pattern in milliseconds.

`<Random seed>` Normally set to `0` which causes the program to create a new seed at each invocation using a modulus-based computation involving measurements of network latencies. Network connectivity for ICMP to the Internet is required. Alternatively, this parameter can be used to set a specific seed value to (re-)create a specific random loss sequence.

`<PVAL>` This parameter has two possible meanings depending on its value. If PVAL is given as a decimal value < 1 then it is used as a packet loss rate (PLR). Each packet position will have a probability equal to the PLR to be set to dropped. Since the loss generation is probabilistic, the exact amount of lost packets in the pattern cannot be exactly controlled. If exact control over the number of lost packets is desired, a PVAL $>=1$ can be specified. The program then interprets this as the number of losses

that should be present in the pattern, and generates a pattern file with exactly this number of randomly placed losses.

Examples:

```
[user@testpc1 prgms]$ patt_gen -pkt -rand testrandpat1.plp data 2400
0 .03
```

This example generates a pattern file where each packet has a probability of 3% of being lost. An internally generated seed is used and the data for the 2400 packet pattern is stored in the file testrandpat1.plp. In this particular run 69 losses were inserted in the pattern file, resulting in an actual loss ratio of 2.87% for the pattern as a whole.

```
[user@testpc1 prgms]$ patt_gen -pkt -rand testrandpat2.plp data 1600
0 16
```

This example generates a pattern file covering 1600 packets with exactly 16 randomly placed packet losses. An internally generated seed is used and the data for the pattern is stored in the file testrandpat2.plp. As specified, 16 losses were inserted in the pattern file, resulting in an actual loss ratio of 1.00% for the pattern as a whole.

C.2.2 Gilbert-Elliot packet loss patterns

The `patt_gen` program can generate pattern files according to the commonly used Gilbert-Elliot model. This model has two states, a good state and a bad state. Each of the states have a packet loss rate (PLR) attached to it, and a transition probability to enter the other state. This model captures time varying loss processes better than the random uniform model.

Syntax:

```
patt_gen -pkt -ge   <filename> <mode> <size> <Random_seed> <Good_PLR>
<Bad_PLR> <Good_tran_prob> <Bad_tran_prob>
```

`<filename>` The name of the output file. The recommended suffix for packet loss pattern files such as generated here is .plp.

`<mode>` Specifies the generation mode (data/time). The mode specification can influence the interpretation of later parameters.

`<size>` The meaning of this parameter is different between the modes. For data-driven it specifies the length of the generated pattern in number of packets. A value of 10000 will generate a pattern covering 10000 packets. For time-driven mode, the size specifies the length of the pattern in milliseconds. A value of 100000 will generate a pattern covering 100 seconds.

<Random seed> Normally set to 0 which causes the program to create a new seed at each invocation using a modulus-based computation involving measurements of network latencies. Network connectivity for ICMP to the Internet is required. Alternatively, this parameter can be used to set a specific seed value to (re-)create a specific random loss sequence.

<Good_PLR> This decimal sets the PLR that applies when being in the good state.

<Bad_PLR> This decimal sets the PLR that applies when being in the bad state.

<Good_tran_prob> This decimal sets the probability of transitioning from the good state to the bad state.

<Bad_tran_prob> This decimal sets the probability of transitioning from the bad state to the good state.

Example:

```
[user@testpc1 prgms]$ patt_gen -pkt -ge testrandpat3.plp time 2048
0 .000000001 .008 .000002 .0008
```

This example generates a packet loss pattern that groups the packet losses in a bursty fashion. The length and interval of the loss bursts are controlled by the transition probabilities. In this example an internally generated seed is used and the pattern is stored in the file `testrandpat3.plp` which covers 2 Mbytes of traffic. In this particular run 425 packet losses were generated.

C.2.3 Fixed position packet loss patterns

In data-driven mode, fixed position patterns allow the specification of which individual packets that are to be lost. For example a pattern losing the 12th, 42nd and 87th packets could be created. In time-driven mode, the patterns control during which milliseconds packet losses will occur. All packets that are scheduled to be sent during this time are lost.

Syntax:

```
patt_gen -pkt -pos  <filename> <mode> <size> <positions>
patt_gen -pkt -pos  <filename> <mode> <size> -f <infilename>
```

<filename> The name of the output file. The recommended suffix for packet loss pattern files such as generated here is .plp.

<mode> Specifies the generation mode (data/time). The mode specification can influence the interpretation of later parameters.

<size> The meaning of this parameter is different between the modes. For data-driven it specifies the length of the generated pattern in number of packets. A value of 10000 will generate a pattern covering 10000 packets. For time-driven mode, the size specifies the length of the pattern in milliseconds. A value of 100000 will generate a pattern covering 100 seconds.

<positions> This is a comma separated list which should list the positions of the packet losses in ascending order, either in absolute position (data-driven) or in time units (time-driven). Counting starts from the value 1 so the use of 0 as a specification results in an error.

-f <infilename> Gives the filename of a text-file that contains packet loss positions or packet loss times. The file should contain an ascending list of loss positions with one value per line without any whitespace in front of the value and no blank lines.

Example:

```
[user@testpc1 prgms]$ patt_gen -pkt -pos testpospat.plp data 20 5,10,15
```

The pattern generated in this example will cause packets number 5, 10, and 15 to be lost as soon as they enter the emulator.

C.2.4 Fixed interval packet loss patterns

Syntax:

```
patt_gen -pkt -int  <filename> <mode> <size> <interval-list> <start_value>
patt_gen -pkt -int  <filename> <mode> <size> -f <infilename> <start_value>
```

<filename> The name of the output file. The recommended suffix for packet loss pattern files such as generated here is .plp.

<mode> Specifies the generation mode (data/time). The mode specification can influence the interpretation of later parameters.

<size> The meaning of this parameter is different between the modes. For data-driven it specifies the length of the generated pattern in number of packets. A value of 10000 will generate a pattern covering 10000 packets. For time-driven mode, the size specifies the length of the pattern in milliseconds. A value of 100000 will generate a pattern covering 100 seconds.

<interval-list> this is a comma separated list which lists the positions of the packet losses, either as absolute position intervals (data-driven) or millisecond intervals (time-driven).

-f <infilename> Gives the filename of a text-file that contains packet loss intervals. The file should contain an ascending list of loss interval positions

with one value per line without any whitespace in front of the value and no blank lines.

`<start_value>` This value decides if the first initial interval should start without losses (`0`) or with losses(`1`). The default is 0.

Example:

```
[user@testpc1 prgms]$ patt_gen -pkt -int testintpat.plp time 80000
5000,5500,22000,23000,67000,69000  0
```

This example generates an 80 seconds time-driven pattern. The provided parameters will result in a pattern file with 500 ones starting at position 5000, 1000 ones starting at 22000 and 2000 ones starting at 67000. This will emulate a 0.5 s outage starting at 5s, and a 1s outage at 22s and a 2s outage at 67s.

C.3 Generating delay change patterns

Delay change patterns can currently only be explicitly specified from the command line or in a file.

C.3.1 Fixed position delay change patterns

In data-driven mode the delay change patterns control from which pattern position a new delay value should be used. For time-driven mode the pattern specifies from which millisecond the new delay value should be used. Note that patterns which contain a sudden lowering of the delay does not lead to packet reordering, as shown in Appendix B.3.2.

Syntax:

```
patt_gen -del -pos  <filename> <mode> <size> <position-value>
patt_gen -del -pos  <filename> <mode> <size> -f <infilename>
```

`<filename>` The name of the output file. The recommended suffix for packet loss pattern files such as generated here is .dcp.

`<mode>` Specifies the generation mode (data/time). The mode specification can influence the interpretation of later parameters.

`<size>` The meaning of this parameter is different between the modes. For data-driven it specifies the length of the generated pattern in number of packets. A value of 10000 will generate a pattern covering 10000 packets. For time-driven mode, the size specifies the length of the pattern in milliseconds. A value of 100000 will generate a pattern covering 100 seconds.

`<position-value>` This is a comma separated list which should contain tuples position-value separated by commas. The first packet/millisecond is numbered 1, so the use of 0 as a position results in an error.

`-f <infilename>` Gives the filename of a text-file that contains tuples of position-value. The expected format is to have a position-value pair on each line separated by comma, but pure comma separation or line separation is also accepted. The positions should be in ascending order.

Example:

```
[user@testpc1 prgms]$ patt_gen -del -pos testpat.dcp data 20 1,50,10,10
```

This example will create a pattern that for the first nine packets sets the end-to-end delay to 50 ms. When the tenth packet enters the emulator, the pattern will decrease the delay to only 10 ms.

C.4 Generating bandwidth change patterns

Just like delay change patterns, bandwidth change patterns can currently only be explicitly specified from the command line or in a file. Note that data-driven bandwidth change patterns should not be used on the same pipe as time-driven bit-errors. Time-driven bit-errors use the bandwidth value to calulate the bit-times, and data-driven bandwidth changes cannot always be adequately compensated for in the bit-time calculations.

C.4.1 Fixed position bandwidth change patterns

In data-driven mode the bandwidth change patterns control from which pattern position a new bandwidth value should be used. For time-driven mode the pattern specifies from which millisecond the new bandwidth value should be used.

Syntax:

```
patt_gen -bw -pos  <filename> <mode> <size> <position-value>
patt_gen -bw -pos  <filename> <mode> <size> -f <infilename>
```

`<filename>` The name of the output file. The recommended suffix for pattern files such as generated here is .bcp.

`<mode>` Specifies the generation mode (data/time). The mode specification can influence the interpretation of later parameters.

`<size>` The meaning of this parameter is different between the modes. For data-driven it specifies the length of the generated pattern in number of packets. A value of 10000 will generate a pattern covering 10000 packets. For time-driven mode, the size specifies the length of the pattern in milliseconds. A value of 100000 will generate a pattern covering 100 seconds.

`<position-value>` This is a comma separated list which should contain tuples position-value separated by commas. The first packet/millisecond is numbered 1, so the use of 0 as a position results in an error.

`-f <infilename>` Gives the filename of a text-file that contains tuples of position-value. The expected format is to have a position-value pair on each line separated by comma, but pure comma separation or line separation is also accepted. The positions should be in ascending order.

Example:

```
[user@testpc1 prgms]$ patt_gen -bw -pos testpat.bcp data 20 1,1000,10,100
```

This example generates a 20 packet long bandwidth change pattern. The provided parameters will result in a pattern that sets the bandwidth to 1000 kbps for the first nine incoming patterns. For the remaining packets, the bandwidth will be decreased to 100 kbps.

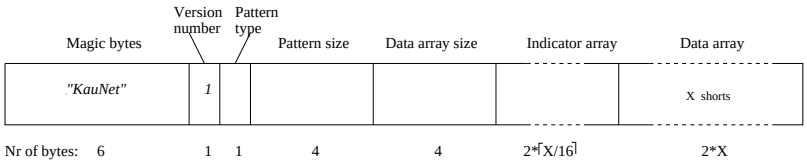


Figure 8: Compressed format

D File formats

D.1 Pattern file format

The pattern files are stored and imported into the kernel in a compressed format. This compressed format is as follows. (All numeric values are unsigned).

- Magic bytes. These numbers are used to verify that the file is a pattern file. The numbers are 75, 97,117, 78,101,116.
- Version number. The only version currently defined is 1.
- Pattern type. This byte is divided into two parts. The two most significant bits indicate if the pattern was produced to be used in time-driven or data-driven mode. The following decimal values for these two bits are defined 1 = Data driven mode, 2= Time driven modem, 3= Data and time driven mode are both applicable (possible for bit-error patterns). The lower six bits encode the pattern types according to the following decimal values: 1=bit errors, 2=packet losses, 3=delays, 4=bandwidth
- Pattern size. This unsigned integer specifies the length of the pattern. The unit of the size is either kilobytes, packets or milliseconds dependent on the type of pattern in the pattern file. The mapping between unit and pattern type is provided in Table 2. Further details are also given in the usage examples, Section C.1 and forward.
- Data array size. This unsigned integer specifies the number of shorts in the data array, which is denoted by X in Figure 8.
- Indicator array. This array of shorts contains indicator values. Each bit in the indicator array indicates if the short at the corresponding position in the data array contains a run-length value or a data value. The number of shorts in the indicator array is$\lceil X/16 \rceil$.
- Data array. This array contains shorts that are either a run-length number indicating the distance to skip (i.e run-length value) until the next short is evaluated, or a data value whose semantics are dependent on the pattern type, as explained in the next subsection.

Pattern type	Size unit for data driven mode	Size unit for time driven mode
Bit-error	Kilobyte	Kilobyte
Packet loss	Packets	Milliseconds
Delay change	Packets	Milliseconds
Bandwidth change	Packets	Milliseconds

Table 2: Size unit

D.2 Data array semantics

The semantics of the data array shorts containing data values are different according to the pattern type and mode as discussed below.

Bit-errors (time or data-driven mode)

For bit-errors the semantics are identical for both modes. The fundamental data value is one bit, which if it is set to one indicates that the bit at the corresponding position in the transferred data should have a bit-error. In this mode each bit in the data value corresponds to one bit that traverses the emulated link.

As mentioned earlier the difference between time-driven and data-driven mode is in how the index into the pattern file is forwarded at emulation time. For time-driven, the index is forwarded at each millisecond as time passes by. The number of bits to forward the index with at each millisecond tick is given by the bandwidth configured for the pipe. If there is data to be sent during this millisecond, this data is subjected to bit-errors as specified in the pattern file. If there is no data to be sent, the index is simply forwarded. In a sense the index is forwarded one bit position for each bit that could have been transmitted. Time-driven bit-errors can be used to in a more repeatable and deterministic way emulate the effects of time-varying channel conditions.

For data-driven bit-errors the index is forwarded one bit for each bit that is transmitted. If no data is transmitted, the index is not forwarded.

Packet losses (data-driven mode)

For data-driven packet losses the fundamental data value is one bit, which if it is set to one indicates that the packet at the corresponding position in the transferred data should be dropped. In this mode each bit in the data value corresponds to one packet that traverses the emulated link. For each packet that traverses the emulated link, the index is forwarded one bit position in the pattern file. This mode thus makes it possible to lose for example the 6th, 29th and 58th packets of a connection.

The data values are interpreted as binary vectors where a 1 signifies a packet loss and a 0 no packet loss. The bit order is MSB to LSB so for data-driven packet losses for example, a loss of the first packet should be encoded as binary 1000 0000 (=128 decimal or 0x80 hex).

Packet losses (time-driven mode)

Also in this mode a 1 in the data value signifies a packet loss and a 0 no packet loss. However, in time-driven mode a one bit in the pattern file means that packets sent during a specific millisecond period should be dropped. Time-driven packet loss can be used to emulate network connectivity outages. By setting 500 consecutive bits to 1 a 0.5 s outage can be created.

Delay changes (time or data-driven modes)

For delay changes the fundamental data value is an unsigned short representing the new delay value. The resolution is 1ms. This means that a range between 0 ms and 65535 ms is possible. This should cover all useful scenarios without the need for any scaling. For time-driven changes the change occurs at some specific millisecond, and for data-driven mode the change occurs after a specific number of packets have been transferred.

Bandwidth changes (time or data-driven modes)

For bandwidth changes the fundamental data value is an unsigned short representing the new bandwidth value. The bandwidth resolution used is 1 Kbit/s. The possible range is 1 Kbit/s to 65.535 Mbit/s. This range should be sufficient for most desired emulation scenarios.[5] For time-driven changes the change occurs at some specific millisecond, and for data-driven mode the change occurs after a specific number of packets have been transferred.

D.3 Scenario file format.

A scenario file is a collection of up to eight pattern files. The scenario header is essentially a pointer to pattern files with in the scenario file, and contains the following fields:

- Magic numbers to identify the file as an KauNet scenario file. The numbers are: 75,97,117,83,99,110.(KauScn).
- Version number, currently only version 1. (1 byte)
- Identification number, to identify and catalog files. (4 bytes)
- 8 offset fields, denotes the offset to patterns in the file from the start of the scenario file. (8x4 bytes)
- comment length, the length of the comment to follow. (2 bytes)
- comment field.

[5] If a need arises to have even higher bandwidths, the code can possibly changed to include a a scaling factor of for example 10, thus changing the range from 10 Kbit/s to 655.35 Mbit/s in steps of 10 Kbit/s.

KauNet: Design and Usage

KauNet is an emulation system that allows deterministic placement of packet losses and bit-errors as well as more precise control over bandwidth and delay changes. KauNet is an extension to the well-known Dummynet emulator in FreeBSD and allows the use of pattern and scenario files to increase control and repeatability. This report provides a comprehensive description of the usage of KauNet, as well as a technical description of the design and implementation of KauNet.

Karlstad University Studies
ISSN 1403-8099
ISBN 978-91-7063-219-8