

KauNet: A Versatile and Flexible Emulation System

Johan Garcia*

Per Hurtig*

Anna Brunstrom*

*Karlstad University, Dept. of Computer Science, SE-65188 Karlstad, Sweden

Email: johan.garcia@kau.se, per.hurtig@kau.se, anna.brunstrom@kau.se

Abstract—This paper presents the KauNet emulation system that provides pattern-based emulation. KauNet enables precise placement of bit-errors, exact and repeatable packet losses, delays and bandwidth variations. The design and performance of KauNet is discussed and an example is also provided of how it can be used to examine a protocol implementation.

I. INTRODUCTION

The performance and behavior of computer networks and protocols can be evaluated at several levels of abstraction ranging from analytical evaluation, via simulation and experiments in an emulated environment, up to full scale live experiments. Each of these approaches provides different trade-offs with regards to the amount of detail considered, model validation requirements, degree of experimental control, reproducibility, and so on. In many cases these approaches are complementary and can provide different insights about the topic under study. In this paper we focus on the emulation approach and present the KauNet emulation system. By employing fine-grained pattern-based control over the emulated behavior, KauNet 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.

The design of KauNet is centered around a number of pattern-handling extensions to the well known Dummynet emulator [11], together with user-space programs for pattern creation and management. The use of Dummynet as a starting point provides a stable codebase that has been in wide-spread use for several years, as well as the integration with the `ipfw` program 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.

With regards to emulating multiple connections or links, KauNet inherits the Dummynet firewall rule capabilities that are specified with `ipfw`. These capabilities make it easy to create multiple emulated flows, emulating for example the effective bandwidths and delays for a set of nodes in a mobile

network. When the nodes move around their effective conditions change, and the conditions in the emulated environment should change over time to reflect this. While Dummynet allows emulated conditions to be changed during the run of the emulation, these changes must be done using command line tools (typically under the control of a script). Using regular Dummynet, there would be a need to precisely synchronize the invocation and execution of the command line tools in order to try and create an exact reproduction from one experimental run to the other. Instead of using command line tools to change the emulated conditions, KauNet uses patterns that describe the desired changes. The patterns are inserted into the KauNet kernel space at the start of the emulation, and allows a much more exact control of when the conditions change. 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 the command line tools to dynamically switch between multiple patterns.

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 (for an example involving web service response times, see [4]).

The rest of the paper is structured as follows. Section II provides some background and discusses related work. Section III presents a more detailed description of the design and capabilities of the KauNet system. Section IV gives an example of how KauNet has been used to examine TCP implementation issues. Finally, some conclusions are provided.

II. BACKGROUND AND RELATED WORK

Network emulation has been used for quite some time in the networking community, and there exists a number of emulation systems. As discussed before, KauNet is an extension to Dummynet [11]. Dummynet is implemented in the FreeBSD kernel, and can be used on users' workstations, or on dedicated PCs acting as routers or bridges. Since Dummynet is integrated with the firewall functionality, packets can easily be classified according to IP address, port number, type of service, and other fields. Matching packets can then be put into different pipes, where different pipes can have their own parameters for bandwidth limitations, delays and loss probability. A similar functionality is provided by NIST Net [1], which allows a Linux PC to emulate a wide variety of network conditions. These include tunable packet delay distributions, bandwidth limitations, packet reordering and packet duplication. Like in

Dummysnet, NIST Net induces non-congestion related losses using a random process. Neither emulator can introduce bit errors. Seawind [5] is designed to study data transfer for a single user, mainly in a cellular environment. Packet loss in Seawind can be based upon various random distributions, but may also be based upon a sequence in an external file.

Trace-driven approaches such as described in [7], [8], [9] measure real traffic and then use the information in the emulated scenario. In [7], probe packets are actively sent to another host, and the delay and loss are measured. In [8], [9], traffic is passively observed and distilled traces can then be used for example in the PaM [8] network emulator. PaM can then drop, corrupt or delay intercepted packets, using the distilled trace as a model representation of the time-varying characteristics of the network.

NCTUns [2] is a hybrid emulator and simulator capable of emulating a wide range of network devices and protocols. Ns-2 [6] is a popular simulator that has been extended with limited emulation capabilities [3]. However it primarily remains a discrete event simulation tool.

A differentiating factor between previous trace-based emulators and KauNet is the level of detail that the traces represent. While most other trace-driven emulators only use traces to calculate the parameters used for stochastic generation of delays and losses, KauNet traces can be used to control the behavior of each individual packet with regards to bit-errors, packet loss, delay, and effective bandwidth. The ability to precisely control the conditions for each individual packet, efficiently and for large numbers of packets, is the major novelty of KauNet.

III. KAUNET OVERVIEW

KauNet is composed of two major parts, Extensions to the Dummysnet code inside the FreeBSD kernel, and tools to create and manage pattern files.

A. Kernel part

KauNet extends the Dummysnet functionality with the ability to precisely position losses, introduce bit-errors, and control delay and bandwidth changes. This new functionality can be precisely controlled on a per-millisecond basis, or a per-packet basis with the use of patterns. 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. This restriction indicates 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.

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 itself is 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.

B. Pattern generation tools

In addition to the kernel pattern management that is handled by `ipfw` as described above, a command line tool has been developed to create and manage patterns. The `patt_gen` 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 models, off-line simulators or trace collection equipment. A detailed description of the `patt_gen` tool can not be provided due to space restrictions. To complement the `patt_gen` tool a GUI called `pg_gui`

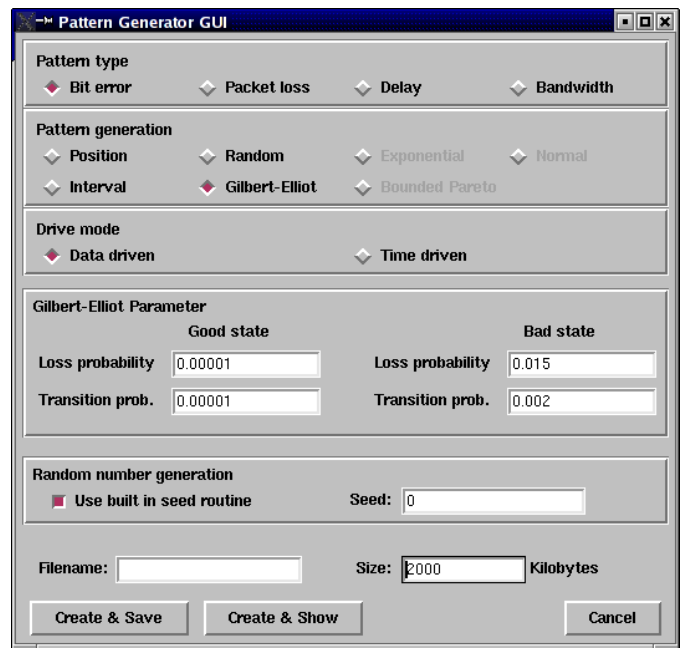


Figure 1. The `pg_gui` pattern creation GUI

has been developed. The appearance of `pg_gui` is shown in Fig 1, where the panels for bit-error pattern generation are shown. From Fig 1 it can be seen 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. Fig 2 shows an example of the pattern generated by the settings shown in Fig 1. 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 a command line tool.

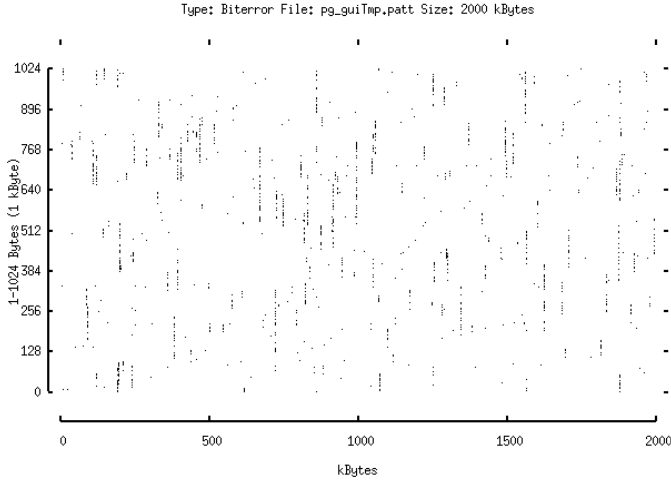


Figure 2. A visualized Gilbert-Elliot bit-error pattern

C. Pattern and scenario files

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.

One pattern is necessary for each of the controllable aspects, i.e. bit-errors, packet losses, bandwidth changes and delay changes. Since many experiments 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 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 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.

D. Emulation performance and scalability

Due to very efficient pattern handling inside the kernel, the performance impact of the KauNet extensions is minimal. Initial experiments performed on a standard desktop computer (2.4GHz Dell OptiPlex GX260, 512 MB RAM) acting as a gigabit Ethernet router shows that the maximum achievable throughput goes down from approximately 368 Mbps (circa 31 kpps¹) when Dummynet/KauNet is not used, to around 352 Mbps (circa 29 kpps) with KauNet enabled and a bit-error pattern with a BER of 10^{-5} . This indicates that the overhead of moving forward in the pattern and applying the bit-errors² requires very little overhead.

With regards to the memory requirements needed to keep the patterns inside the kernel, it should be noted that the patterns are stored using a run-length compressed format, and that decompression occurs stepwise as the pattern information is consumed. As an example, take the storage requirements needed for the 2 MB bit-error pattern shown in Fig 2. That specific pattern contains 1483 bit errors resulting in a BER of $9.1 \cdot 10^{-5}$. As an uncompressed bit-pattern this pattern would need 2 MB, expressing the loss positions as a textual list requires 12285 Bytes, and as a compressed pattern the space required is 5234 Bytes. The actual storage requirement for a pattern is dependent on the type of pattern and the amount of bit-errors, losses, etc. in the pattern. We have successfully imported patterns over 20000 times the size of the example pattern described earlier (i.e. > 100 MB). This fact, in conjunction with the ability to seamlessly integrate a new pattern when the current pattern is ended, minimizes the risk for scalability problems due to the pattern data requirements of KauNet.

Considering that the FreeBSD kernel is not a real-time kernel, the exact timing of imposed packet delays cannot be guaranteed if the system is overloaded. In the experimental work performed so far with KauNet, this has not been a noticeable problem. However, most of this work has not been performed at heavy loads. Further examination of KauNet when emulating high bandwidths and at extreme loads are part of our current work.

IV. EVALUATION EXAMPLE

The deterministic loss placement functionality provided by KauNet can for example be used to evaluate transport protocol parametrizations and to verify the correct functioning of protocol implementations. In this example, experiments are performed on the TCP implementation in FreeBSD 6.0. These experiments focus on short 25 KB connections where the transient behavior has a large impact³. The ability of KauNet to place packet losses at precise positions is used to examine how the total transfer time is affected by packet losses in the

¹kpps=kilo packets per second.

²on average, there is one bit-error in approximately every 8th packet.

³For the used MTU of 1500 Bytes, this will result in a total of 20 packets sent from the server.

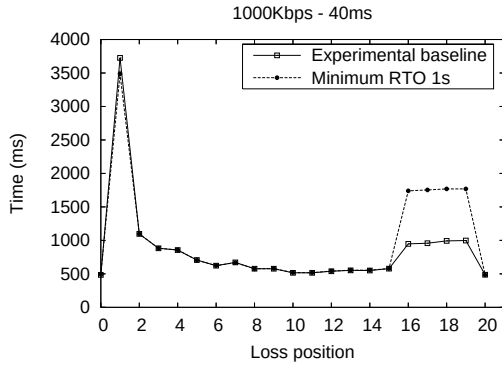


Figure 3. The effect of the setting minRTO to 1 s for 1000 Kbps - 40 ms

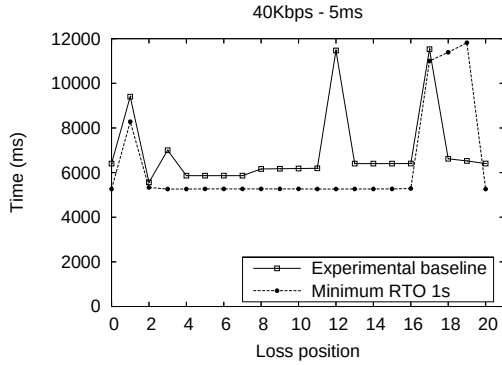


Figure 4. Unexpected problem for 40 Kbps - 5 ms due to opportunistic RTO calculations

direction from the TCP sender. An example result is shown in Figure 3 which shows how the transfer time is affected by packet losses at different positions. Two runs are shown in the figure, one using the baseline parametrization which uses a minimum retransmission time out (minRTO) of 200 ms. For the second run the the minRTO is set to 1 s, which is the value recommended in [10]. As expected the figure shows a peak at loss position one which corresponds to losing the SYNACK segment. There is an additional bump in transfer time toward the end of the connection. For the segments at the end of the connection TCP must use a timeout to recover from a packet loss since there are not enough data left to generate three duplicate acknowledgments. Since these losses are recovered by timeout, the effect of an increased minRTO is clearly visible for loss positions 16-19.

While the results shown in Figure 3 matches what could be expected, other experiments highlighted some weaknesses in the examined implementation. The results for the experimental runs performed with a smaller bandwidth and a smaller delay are shown in Figure 4. In this figure the baseline is slower than the minRTO=1 case in many instances. Further examination of this issue with additional KauNet experimental runs and code inspection showed that this effect is due to an opportunistic RTO calculation technique employed in FreeBSD 6.0. Instead of using an initial RTO value of three seconds [10] for the data that is sent in the first window, FreeBSD uses the acknowledgments in the connection establishment phase

to calculate an RTO value. This can be seen as a rather natural optimization of the RTO calculation, but there are some drawbacks with this strategy. Let us consider a client that connects to a server and the path that they communicate over has a bandwidth of 40 Kbit/s. The round-trip time of the SYNACK-ACK segments that TCP uses while establishing the connection will be approximately $\frac{78 \text{ Bytes} \cdot 8 \text{ bits/Byte}}{40 \cdot 10^3 \text{ bps}} + \frac{66 \text{ Bytes} \cdot 8 \text{ bits/Byte}}{40 \cdot 10^3 \text{ bps}} = 0.0156 \text{ s} + 0.0132 \text{ s} = 0.0288 \text{ s}$, if we omit the link delay⁴. If we calculate the RTO based on this round-trip time according to the kernel source code, it will be approximately 0.287 s. If we then consider the round-trip time of a full sized (1500 Bytes) segment we get approximately $\frac{1500 \cdot 8}{40 \cdot 10^3} + \frac{66 \cdot 8}{40 \cdot 10^3} = 0.3132 \text{ s}$. Thus, for low link delays the initial window of data sent from the server to the client will always time out and be spuriously retransmitted, causing a severe performance penalty on the connection. This is the reason behind the anomalous behavior found in Figure 4. Due to the spurious retransmissions more than 19 packets are needed for the data transfer so the end-of-connection bump is in effect pushed out to the right of Figure 4. Interaction effects are also created as the RTO timer will back-off exponentially for each timeout that is experienced, thus making error recovery very expensive if a subsequent packet is lost.

V. CONCLUSIONS

The KauNet trace-based emulation system has been developed to allow fine-grained and repeatable control of bit-errors, packet losses and packet delays. The use of scenario files and a central scenario repository simplifies the reproduction of experiments and encourages sharing.

REFERENCES

- [1] M. Carson and D. Santay. NIST Net: A linux-based network emulation tool. *ACM SIGCOMM Computer Communication Review*, 33(3):111–126, 2003.
- [2] S. Y. Wang et al. The design and implementation of the nctuns 1.0 network simulator. *Computer Networks*, 42(2):175–197, June 2003.
- [3] K. Fall. Network emulation in the VINT/NS simulator. *Proceedings of the fourth IEEE Symposium on Computers and Communications*, July 1999.
- [4] J. Garcia, P. Hurtig, and A. Brunstrom. The effect of packet loss on the response times of web services. In *press for 3rd International Conference on Web Information Systems and Technologies (WebIST2007)*, Barcelona, Spain, March 2007.
- [5] M. Kojima, A. Gurtov, J. Manner, P. Sarolahti, T. Alanko, and K. Raatikainen. Seawind: A wireless network emulator. In *Proceedings of 11th GI/ITG Conference on Measuring, Modelling and Evaluation of Computer and Communication Systems*, September 2001.
- [6] S. McCanne and S. Floyd. VINT Network Simulator - ns (version 2). <http://www.mash.CS.Berkeley.EDU/ns/>, April 1999.
- [7] B. Melander and M. Björkman. Trace-driven network path emulation. Technical report 2002-037, Department of Information Technology, Uppsala University, Sweden, 2002.
- [8] B. Noble, G. Nguyen, M. Satyanarayanan, and R. Katz. RFC 2041: Mobile network tracing, October 1996.
- [9] B. Noble, M. Satyanarayanan, G. Nguyen, and R. Katz. Trace-based mobile network emulation. In *Proceedings of ACM SIGCOMM '97*, September 1997.
- [10] V. Paxson and M. Allman. RFC 2988: Computing TCP's Retransmission Timer, November 2000.
- [11] Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, January 1997.

⁴The SYN_ACK and ACK segments differ in the number of TCP options carried and have the sizes 78 and 66 Bytes respectively.