Faculty of Economic Sciences, Communication and IT
Department of Computer Science

Tomas Hall Andreas Midestad

KauNet Triggers

Degree Project of 30 ECTS credit points
Master of Science in Information Technology

Date/Term: 10-01-22
Supervisor: Per Hurtig
Examiner: Donald F. Ross
Serial Number: E2010:02

Karlstads universitet 651 88 Karlstad
Tfn 054-700 10 00 Fax 054-700 14 60
Information@kau.se www.kau.se

KauNet Triggers

Tomas Hall **Andreas Midestad**

© 2010 Tomas Hall, Andreas Midestad and Karlstad University

This thesis is submitted in partial fulfillment of the requirements for the Masters degree in Computer Science. All material in this thesis which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Tomas Hall

Andreas Midestad

Approved, 2010-01-22

Opponent: Johan Nordholm

Advisor: Per Hurtig

Examiner: Donald F. Ross

Abstract

An important aspect of development and research in the field of computer networking systems is evaluation. Through evaluation, performance and behavior of software and protocols over a network can be determined. A network emulator is one of several tools available to accomplish this.

In this thesis, the network emulator Dummynet is described, as well as its extension KauNet. KauNet extends Dummynet by introducing pattern-driven emulation. A pattern defines specific points at which to apply a certain computer network characteristic or behavior. The use of patterns allow an increased control and repeatability of an emulation. Repeating a test with an identical configuration and the same pattern will yield identical results.

The goal of the project was to add a new functionality to KauNet. The new functionality consists of a notification system capable of passing information from KauNet to external observers. By adding this new functionality, emulation statistics can be available for the observers immediately when occurring. Another example of information that can be forwarded, is simulated cross-layer information. For KauNet to know when and what information to send, a new type of pattern has been created, called trigger pattern. Trigger patterns behave similarly to the existing patterns, sharing the same structure and processing in KauNet. Through the use of trigger patterns, events may be raised at specific points. The notification system may then be used to pass the event information.

This thesis describes the evaluation, design and implementation of the trigger patterns

and notification system in KauNet. Finally, it concludes with a verification of the new trigger functionality in a usage example.

We would like to thank our supervisor Per Hurtig for all his invaluable help.

Contents

1 Introduction **1**

1.1 Problem, Goals & Motivation . . . 2

1.2 Disposition . . . 3

2 Background **5**

2.1 Evaluation of Computer Networking Systems . . . 5

2.1.1 Theoretical Analysis . . . 6

2.1.2 Simulation . . . 6

2.1.3 Emulation . . . 8

2.1.4 Live Testing . . . 10

2.2 Dummynet . . . 11

2.2.1 IPFW & Packet Classifying . . . 13

2.2.2 Pipes & Queues . . . 14

2.2.3 Dummynet Usage Example . . . 15

2.3 KauNet . . . 17

2.3.1 Patterns . . . 18

2.3.2 Pattern Generation Utility . . . 22

2.3.3 KauNet Usage Example . . . 23

2.4 Summary . . . 24

3 Problem Description **27**

3.1 Background 27

3.2 Motivating Examples 28

3.2.1 Real-Time Emulation Updates 28

3.2.2 Cross-layer Optimization 29

3.2.3 Link Properties Estimation 31

3.3 Design Considerations 31

3.4 Summary 32

4 Design **33**

4.1 Triggers 33

4.2 Trigger Passing 34

4.2.1 KauNet 35

4.2.2 Trigger Passing Mechanism Evaluation 35

4.2.3 KauNet Communication Module 47

4.2.4 Adaptation Layer 49

4.2.5 Adaptation Layer Communication Module 49

4.3 Summary 50

5 Implementation **53**

5.1 Dummynet/KauNet Architecture 53

5.2 Trigger Patterns 57

5.2.1 Pattern Structure 57

5.2.2 Pattern Generation 59

5.3 Trigger Passing 60

5.3.1 KauNet 60

5.3.2 KauNet Communication Module 61

5.3.3 Adaptation Layer Communication Module 62

5.3.4 Adaptation Layer 64
5.4 Summary 64

6 System Demonstration **67**
6.1 Theory 67
6.2 Method 69
6.3 Results 71
6.3.1 Experiment 1 71
6.3.2 Experiment 2 72
6.4 Conclusions 73

7 Conclusions & Future Work **75**
7.1 Future Work 76

References **77**

A IPC Evaluation **81**
A.1 Test Machine Specifications 81

B FreeBSD **83**

C Source code **85**
C.1 File Difference Output 85
C.2 FreeBSD Kernel (with KauNet) 86
C.2.1 ip_dummynet.h 86
C.2.2 ip_dummynet.c 87
C.3 KauNet Communication Module 91
C.3.1 kcm.h 92
C.3.2 kcm.c 94
C.4 Adaptation Layer Communication Module 101

C.4.1 alcm.h 101
C.4.2 alcm.c 103
C.5 Adaptation Layer 106
C.5.1 al.c 107
C.6 Pattern Generation Utility 109
C.6.1 patt_gen.diff 109
C.7 The Internet Protocol Firewall 116
C.7.1 dummynet.c 116

List of Figures

2.1 Simulator. 6

2.2 Emulator. 8

2.3 Live testing. 10

2.4 Dummynet - Emulation overview. 12

2.5 Dummynet pipes and traffic filtering. 13

2.6 KauNet - Emulation overview. 17

2.7 Example 1 - Dummynet with 50% packet loss. 18

2.8 Example 2 - KauNet with 50% packet loss. 18

2.9 Data-driven packet loss example [19]. 21

2.10 Time-driven packet loss example [19]. 21

3.1 KauNet Trigger Example. 29

4.1 Passing triggers using an IPC mechanism. 36

4.2 Passing triggers using network sockets. 36

4.3 Raw signal data transfer. 38

4.4 Dynamic shared memory data transfer. 39

4.5 Static shared memory data transfer. 39

4.6 Maximum reliable rate as a function of the payload size. 43

4.7 Maximum bandwidth as a function of the payload size. 44

4.8 KauNet communication module using an intermediate receiver and IPC. . . 47

4.9 KauNet communication module using sockets. 48
4.10 KauNet trigger passing. 50

5.1 Dummynet event flow. 54
5.2 KauNet pattern structure (compressed format). 58
5.3 KauNet trigger passing implementation overview. 65

6.1 Test bed setup. 67
6.2 Experiment 1 setup. 69

List of Tables

2.1 Pattern behavior in data-driven mode. 19
2.2 Pattern behavior in time-driven mode. 19

5.1 Type of pattern and unit mapping. 59

6.1 Theoretical experiment results. 68

A.1 IPC test bed specification, KauNet host. 81
A.2 IPC test bed specification, receiving host. 82

Chapter 1

Introduction

Computer networking is an ever expanding field of development and research, especially since the advent of the Internet (that is, once people realized that it was, in fact, not a fad). Over the years, countless programs and protocols have been developed to provide a wide range of services and functionality for users and providers alike. As the number of applications and the size of the networks grow, so does the need to evaluate new and existing network systems. One of the methods available is the use of a network emulator; a piece of software used to simulate characteristics of different network setups, without the need for a real network using the actual hardware components.

One well known and commonly used emulator available for several platforms is Dummynet (see Section 2.2). Dummynet is capable of emulating complex network topologies and apply a variety of emulation effects such as packet losses and bandwidth restrictions on traffic that passes through it. One significant drawback is the lack of a reliable way to reproduce results in experiments performed. This led to the eventual development of KauNet, an extension built on top of Dummynet (see Section 2.3). In addition to providing all of the functionality of Dummynet, KauNet introduces a new concept; deterministic emulation effects using a pattern based system. Patterns are data files inserted into KauNet, defining specific points at which to apply different emulation effects. Each pattern con-

sists of a number of values divided into discrete steps, each step representing either one millisecond or one packet in the emulation, depending on the mode of operation. Each value represent an emulation effect to be applied at the specified point. Through the use of patterns, experiments can be repeated with a high degree of reproducibility by using the same patterns and traffic.

1.1 Problem, Goals & Motivation

Currently, KauNet statistics and limited emulation information is only available on demand. The purpose of this thesis is to introduce a new mechanism (*trigger passing*) to allow observers residing outside the emulator (*subscribers*) on both local and remote machines to receive notifications (*events*) from KauNet. With the ability for KauNet to send notifications automatically and in real-time, it is possible to emulate additional network characteristics, such as cross-layer information. It can also be used for more general updates and control messages. Examples of the use of notifications in KauNet are given in greater detail in Section 3.2.

The trigger passing functionality is to be realized by implementing a new type of KauNet pattern; the *trigger pattern*. The trigger pattern should be implemented in such a way that it can be used in the same manner as, and synchronized with, the existing patterns. This allows notifications to be sent at specific points in time with the same granularity as other KauNet patterns. Each time the trigger pattern is invoked, KauNet should send an event notification out of the emulator, including any information deemed necessary for the receiving subscriber. The goals of this thesis are to;

1. Determine different methods for how the trigger passing mechanism can be implemented.

2. Determine what and how much information the different methods can send.

3. Evaluate the methods to ensure they perform sufficiently well.

4. Choose an appropriate method and implement the trigger passing mechanism.

5. Implement a way to generate the trigger patterns.

6. Modify KauNet to support trigger patterns and make use of the trigger passing mechanism to send out event notifications.

The minimum requirements for the trigger passing are that an event should contain at least four bytes of information, corresponding to the trigger value specified by the content of the loaded trigger pattern. The mechanism should also be able to send events reliably (without losses) at a rate equal to or greater than the update rate of the time-driven KauNet patterns (i.e. one event per millisecond).

1.2 Disposition

This report consists of a total of 7 chapters plus an appendix.

The background, ideas and programs used in the thesis are described in Chapter 2. It provides an overview of different methods that may be used when evaluating the performance of computer network systems. It also introduces the network emulator Dummynet and its extension KauNet, describing what they are, how they can be used and general functionality, as well as a simple usage example.

The thesis project is introduced in Chapter 3. It explains the problem to be solved, motivating why and how trigger patterns and trigger passing can be used to solve this. The considerations that had to be taken into account during the initial design phase and evaluation are also mentioned here.

The overall system design is described in Chapter 4. It describes the trigger concept and trigger patters in greater detail, what they are and what they do. It explains the purpose and design of the trigger passing mechanism, motivating the design with an evaluation

of different possible implementations of the mechanism with an experimental prototype trigger passing mechanism. Based on this evaluation, the design and purpose of the components that are to be implemented for the trigger passing mechanism are explained and motivated.

Chapter 5 describes the implementation of the entire system. It describes the Dummynet/KauNet architecture and implementation in detail as a background for the implementation of triggers and the trigger passing mechanism. The structure and creation of trigger patterns is explained, as well as the implementation details of the trigger passing mechanism using the components introduced in Chapter 4.

Chapter 6 provides a usage example of how trigger patterns can be generated and used with KauNet in the form of a simple demonstration experiments. In addition, it shows how custom adaptation layers can be designed that makes use of the implemented functionality. The experiments also serves as a verification for the implementation.

The conclusions of the thesis are in Chapter 7. It summarizes the project purpose, design and implementation, and a short evaluation of the results. Finally, it includes possible topics for future work and improvements to the system.

Chapter 2

Background

Theoretical analysis, simulation, emulation and live testing are all different methods to evaluate performance and behavior of computer networking systems. This chapter gives a brief overview of the mentioned methods, as well as an introduction to KauNet, an extension to the widely used emulator Dummynet, the emulator used in this thesis. A description of the Internet Protocol Firewall application (IPFW), which is used to configure KauNet, is also available. There are also a few examples provided to illustrate the setup and usage of KauNet and pattern creation.

2.1 Evaluation of Computer Networking Systems

Evaluation of computer networking systems is often performed to determine the performance and behavior of software (for example a network protocol) over a network. To get an estimate of how different software behaves, there are many different ways to test it. In the following sections, the four primary methods available are described for how evaluation of software over a network can be performed. The methods include theoretical analysis, simulation, emulation and live testing.

2.1.1 Theoretical Analysis

The goal of doing a theoretical analysis of a computer networking system is to understand and control how network and traffic are formed. The understanding helps when designing mechanisms and algorithms to efficiently locate and share information. Mathematical models can be used to understand the performance limits and trade-offs in computer networks. For example, web browsers and other network related applications use the Transmission Control Protocol (TCP) [1] to transfer information between computers over the Internet. As there are many applications relying on TCP, it is a massive candidate for theoretical analysis. An example of this is [2], where they more accurately predict TCP send rate using a simple analytic characterization.

2.1.2 Simulation

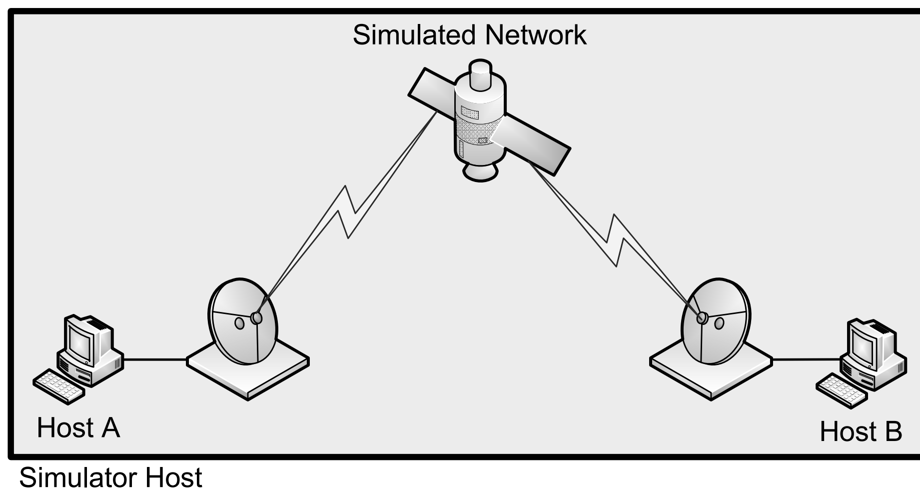

Figure 2.1: Simulator.

A network simulator is essentially a software implementation of a real network, used to mimic the expected behavior of a network in a given scenario. Rather than using the actual hardware, all components in the simulator, such as routers, hosts and links are virtual, creating a representation of a real network and its components. As such, for a complete

and accurate simulation, each component and protocol used in the simulation must be implemented separately, along with any effect they may have on the traffic. An example of a simple network simulator setup is illustrated in Figure 2.1, featuring a simulation of a satellite connecting two end hosts. In addition to the hosts, dishes, satellite and links connecting them, a traffic generator component would also have to be implemented in order to simulate the traffic between the two hosts. By using a software implementation, testing becomes hardware independent and with a great deal of control, offering a number of benefits.

First, the level of control over each component allows each step in the simulation to be monitored closely, allowing full control of anything taking place within the simulator. This makes it easy to identify potential bottlenecks and problems in the implementation. In addition, it offers easily reproducible results by simply using the same simulation parameters to repeat the test [3]. Second, as the testing is done in a virtual environment, no special hardware is required. A simulator can typically run on a single work station. This is especially beneficial in scenarios with large or advanced networks, where creation and modification to the experiment may be both expensive and complicated to perform [3], e.g. an Internet scenario. Eliminating the need to build an actual network also means the test scenarios are easy to both create and alter by simply adding or removing components to the virtual environment. Last, the hardware independence means that the simulation is not subject to limitations found in real hardware and software implementations. A simulator can easily ignore these limitations, such as protocol overhead or routing delays, or even offering features that are not possible in real systems, such as links with extremely high bandwidth (and traffic generators capable of taking advantage of it) or free of corruption. This is possible as the simulator does not have to run in real-time or transfer any actual data over a network link.

Unfortunately, the software implementation also has drawbacks. A real network with real protocols and hardware and software components contain numerous factors that may

influence the traffic and test results in ways that are hard to predict. Implementing all these factors in the simulator may be difficult or even impossible (cf. [4][5][6]). Therefore, it is important to note that the accuracy of any achieved results depend greatly on the level of abstraction and how accurate the implementation of the simulator is. As a result, many simulators either have a high level of abstraction or are specialized for certain scenarios.

Given these properties, simulators are well suited for performing an initial evaluation of a system as no potentially expensive prototype needs to be built. For this reason it is often popular within the academic community, and numerous simulators are available. However, evaluation using a simulator should be seen as complementary to more thorough testing using the actual implementations as opposed to a simulator implementation, in order to obtain more accurate results. An example is the network simulator 2 (ns-2) [7] and its successor ns-3 [8] (currently in development).

2.1.3 Emulation

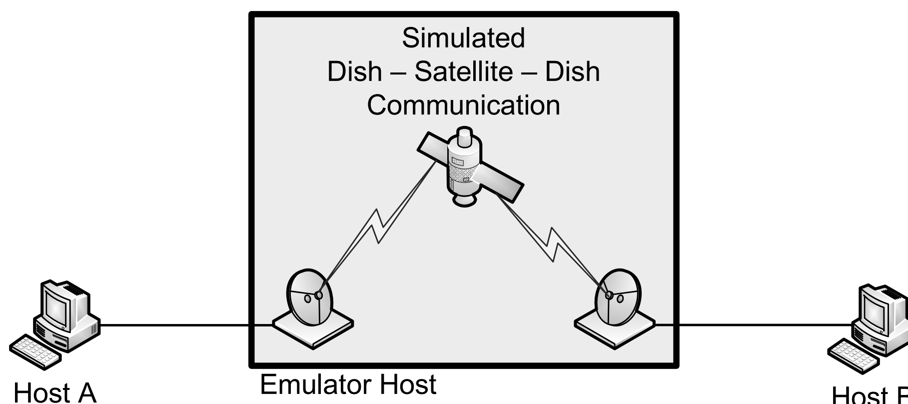

Figure 2.2: Emulator.

An alternative to simulation is emulation. In practice, an emulator is fairly similar to a simulator in that it simulates a network rather than using an actual one. The main difference is that it also uses real hardware and software in the experiment to an extent. This difference is illustrated in Figure 2.2, using the same experiment setup as with the

simulator (see Figure 2.1 for comparison). Note the difference in the setup, where the emulator uses two real hosts connected via an emulator host, whereas the simulator uses virtual hosts. This is a typical setup for network emulators. The hosts themselves provide the traffic generation functionality (using a specialized tool such as Iperf [9], or an arbitrary application), as well as handle any end-to-end functionality. The emulator host simulates the intermediate network characteristics. This is done by intercepting the incoming and outgoing traffic sent by the end hosts and applying various emulation effects on it, such as dropping, delaying or modifying the content of the packets.

Like a simulator, an emulator avoids the reliance on any actual hardware beyond the hosts and network link to emulate the characteristics of a specific network configuration. Therefore, setting up and altering an experiment is fairly easy, for the most part consisting of (re)defining the emulation effects. An important difference is that an emulator uses the actual implementation of protocols, software and some of the network hardware (as opposed to the simulator, which uses its own implementation). For this reason, it is sometimes referred to as a *real-time simulator*. Emulators generally only simulate an end-to-end effect on the traffic [10], imposing the characteristics of the network on the traffic, whereas a simulator generally tries to create a virtual representation of the entire network topology. This increases the level of abstraction, reduces the control over the environment and makes it considerably more difficult to identify specific problems in the test when compared to a simulator, as the emulator is essentially a black box. However, increased abstraction also reduces complexity and offers a number of advantages. This solution means that the emulator needs to implement less functionality, making the implementation less complex and generally allowing emulators to offer broader capabilities than a simulator, though also less detailed. In addition, the reliance on real implementations means that emulators are subject to the same inherent limitations as the original implementations and able to automatically include some of the factors that a simulator may find difficult to implement, bringing the tests closer to a real scenario. As a bonus, code implemented

for testing using an emulator can often be reused as-is in the final product, whereas a simulation implementation may require a complete rewrite. A drawback from the reduced control comes in the form of a somewhat lower degree of reproducibility due to the use of components outside the scope of the emulator, introducing additional factors that cannot be fully controlled. Use of deterministic emulation effects greatly reduces this problem however, offering fairly easily reproducible results.

An emulator is a compromise between a simulator and a real network, taking advantage of properties from both; the repeatable, controllable and easily created environment of a simulator, with the use of real code and hardware from a real network for more accurate results. Numerous emulators are available, such as the previously mentioned ns-2 and ns-3 with its provided emulator interface, as well as Dummynet [11] and its extension KauNet [12], both of which will be explained in greater detail later in this chapter.

2.1.4 Live Testing

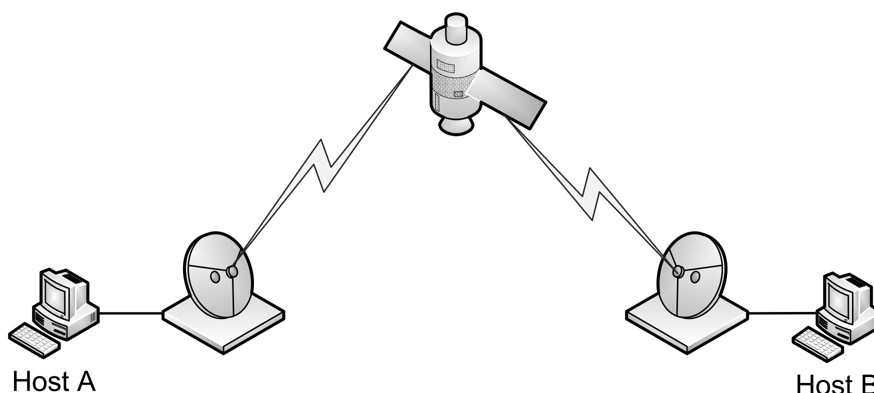

Figure 2.3: Live testing.

Finally, the most direct method of evaluation is to use a real network, such as the Internet, or a test bed for more specific tests. The most immediate disadvantage should be obvious; it requires an actual network with all its components to be constructed and configured. The same scenario as in the previously mentioned simulation and emulation

examples (see Figure 2.3) would require a setup involving an actual satellite, which is generally infeasible for testing purposes. Live testing over custom networks is therefore potentially expensive, both in terms of hardware costs and time, and difficult to modify. An alternative is to use existing infrastructure, such as the Internet, but that option raises another problem with live testing. Reproducing test results may be difficult or even impossible due to the large number of factors beyond the control of the experiment. For instance, the traffic pattern on the Internet changes constantly and the conditions are unlikely to be identical between tests. Even controlled test beds can experience changing conditions as random events in the nodes due to software or hardware may influence traffic generation or forwarding. Furthermore, it may be difficult to accurately interpret the results. An unexpected result could be caused by the tested system itself, but also by the implementation of the transport protocol, operating systems or hardware used in the test bed [3]. The use of the actual protocols and hardware also means that all factors are automatically included, unlike simulators and emulators where some or all of these must either be ignored or implemented explicitly [3][6]. This in turn means that live testing is usually a far more accurate way to estimate the performance of a system and its behavior in a real world scenario. Live testing is therefore an important part of the evaluation, despite its difficulty to deploy and use.

2.2 Dummynet

One network emulator widely used is Dummynet [5], originally developed by Luigi Rizzo in the late 1990s to run network experiments. It has since then also been used for traffic shaping and bandwidth management. The emulator is a software implementation and, for the last decade, a standard component of FreeBSD (see Appendix B). It is also available for Mac OS X (since 2006), and a ported version for Linux was recently released (2009). Dummynet supports various emulation features, including bandwidth limitations, delay,

and probabilistic packet losses over multiple links. Multi-hop links can be emulated by reinjecting traffic into the emulator, allowing Dummynet to be used to emulate both simple and complex topologies. The emulator can be further extended by simply linking several instances of Dummynet or physical systems in the test bed.

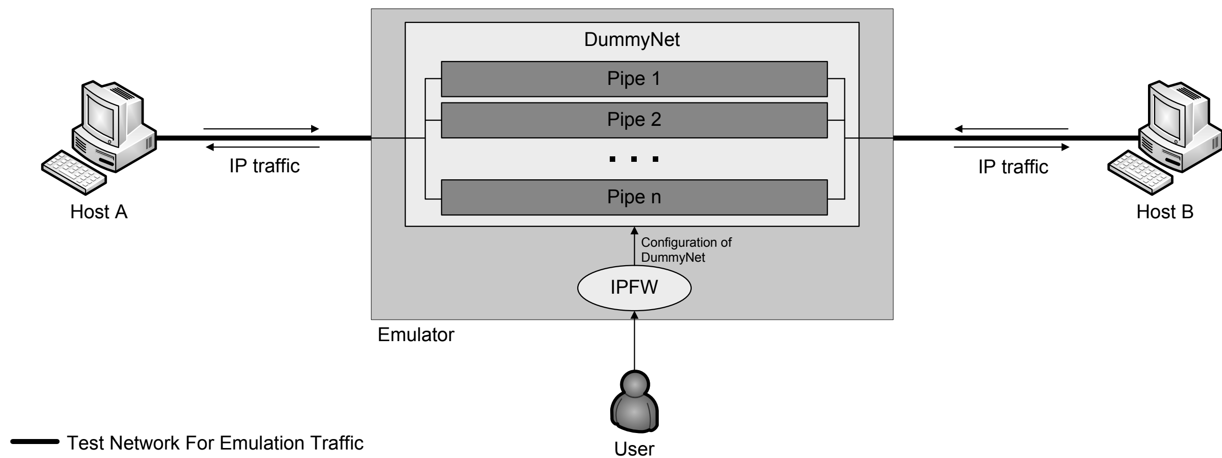

Figure 2.4: Dummynet - Emulation overview.

An example of an emulation setup using Dummynet is shown in Figure 2.4. Host A and Host B are connected to each other through the Dummynet Network Emulator, forming a dedicated experiment network. These hosts are used as the source and destination for the traffic in the experiment and they also generate the traffic itself, while the Dummynet machine emulates the intermediate network. The user interacts with Dummynet via the IPFW application (see Section 2.2.1), which creates and manages the pipes and queues used in Dummynet (see Section 2.2.2). In addition to providing the interface for Dummynet, IPFW also manages the rules used for packet classifying. Each pipe represents a flow and the rules can be used to match traffic against certain conditions and filter it through specific pipes. Dummynet works by intercepting network traffic in the protocol stack and applies the specified emulated effects for each pipe, for example bandwidth limitations [13].

Figure 2.5 illustrates in detail the path of a packet through the emulator. As a packet enters the emulator from the network stack, it is classified by IPFW by matching it to one

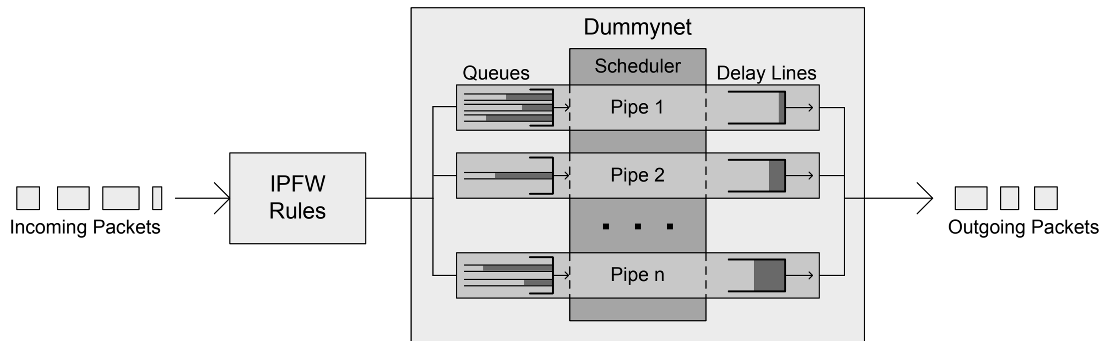

Figure 2.5: Dummynet pipes and traffic filtering.

of its predefined rules. The packet is then placed in one of the queues belonging to the pipe specified by the matched rule. Alternatively, if packet dropping has been configured for the pipe and is triggered by the packet, the packet is immediately dropped. Dummynet drain these queues at a rate corresponding to the bandwidth of the emulated link, using one of several possible queue management schemes, including Random Early Detection (RED) [14], First-In-First-Out (FIFO) and Worst-case Weighted Fair Queuing (WF^{2}Q+) [15][16]. As the queues are drained, the packet eventually enters the scheduler. The scheduler processes the emulated effects of the link to calculate the packet's expected output time. Based on this, the packet is then placed in a delay line, from which it is eventually removed and reinjected into the network stack.

2.2.1 IPFW & Packet Classifying

The Internet Protocol Firewall (IPFW) [17] is an optional firewall application part of and sponsored by FreeBSD. It is used to control the traffic flow through the use of rules created through a built-in stateless rule syntax. With the addition of Dummynet, IPFW also provides an interface for interacting with the emulator, and support is added for the pipe and queue objects used in Dummynet. Through IPFW, additional pipes and queues can be added to and removed from Dummynet, existing pipes and queues can be configured, and traffic flow statistics can be retrieved. It also provides the packet classification functionality

used to divide traffic into the various pipes and queues in Dummynet.

IPFW filters traffic based on a rule hierarchy created by the user. When a packet enters or leaves the host, IPFW attempts to match it to a rule. These can include, for instance, the type of protocol used and the source or the destination (IP address or port number). Several similar rules may be added, which means that traffic may match more than one rule. In these cases, IPFW will attempt match the traffic to the rule with the highest priority (the lowest rule ID number) and work its way through the rules by decreasing priority until it finds one that matches. Although by default, IPFW will only match a packet once to a single rule, it is also possible to reinject matched packets into the classifier. When matching reinjected traffic, IPFW will continue matching from the last matched rule in order to prevent infinite loops. Once a matching rule has been selected, IPFW can choose how to handle the traffic, for example by allowing or denying it. By default, IPFW contains a rule with the lowest possible priority, matching any traffic that has not already been matched, that simply drops the packet.

2.2.2 Pipes & Queues

A pipe is essentially a representation of a traffic flow, along with certain restrictions and modifiers placed upon it. These restrictions provide the emulated properties of the link, such as maximum bandwidth, packet losses, or any other effects Dummynet is capable of emulating. Dummynet can use an arbitrary number of pipes, each identified by a unique identifier and configured individually with any combination of emulation effects.

Queues are queues of packets that share the bandwidth of a single pipe. Several queues can be attached to a single pipe, which also contains a default queue (meaning a queue does not need to be explicitly defined for each pipe). As with pipes, they can be configured separately, to accept a certain type of traffic or different sizes. The Dummynet scheduler uses one of several queuing disciplines to drain the queues as fast as possible, as determined by the emulated bandwidth of the pipe.

As illustrated in Figure 2.5, the rules in IPFW is used to filter the traffic through specific pipes. This is done by allowing IPFW to create rules with pipes attached to them as the action taken for a matched rule. One rule may for example match only TCP traffic and divert this type of traffic through a specific pipe. Another rule may match UDP traffic and filter this through another pipe. Using different rules and pipes, arbitrary topologies of pipes can be constructed as needed [13].

2.2.3 Dummynet Usage Example

Below is a simple example illustrating how probability based packet loss, one of the emulation options in Dummynet, can be configured and used. The test bed is setup as in Figure 2.4, where host A (10.0.1.1) attempts to ping host B (10.0.2.1) via the Dummynet host.

```
Emulator# ipfw add 1 pipe 42 icmp from 10.0.1.1 to 10.0.2.1 out
```

Using IPFW on the KauNet host, a rule with ID 1 (highest possible priority) is added. The action for this rule is to filter the traffic through a pipe, with ID 42. The type of traffic that matches this rule is any traffic using the Internet Control Message Protocol [18] (ICMP, e.g. ping), from 10.0.1.1 to 10.0.2.1, applying to outgoing traffic only.

```
Emulator# ipfw pipe 42 config bw 10Mbit/s delay 10ms plr 0.3
```

Next, IPFW is used to configure pipe 42 with a set of emulation parameters. The pipe receives a bandwidth limitation of 10 Mbit/s as well as a delay of 10 ms (in addition to any normal delay). Finally, a packet loss rate of 30% is set.

```
Host A# ping -i 0.1 -c 20 10.0.2.1
```

Illustrating the effects on the emulated link, host A attempts to ping host B (in this example using 20 packets at a rate of 10 per second).

```
PING 10.0.2.1 (10.0.2.1): 56 data bytes
64 bytes from 10.0.2.1:  icmp_seq=0 ttl=64 time=10.771 ms
64 bytes from 10.0.2.1:  icmp_seq=1 ttl=64 time=11.713 ms
64 bytes from 10.0.2.1:  icmp_seq=2 ttl=64 time=11.299 ms
64 bytes from 10.0.2.1:  icmp_seq=6 ttl=64 time=11.956 ms
64 bytes from 10.0.2.1:  icmp_seq=8 ttl=64 time=11.285 ms
64 bytes from 10.0.2.1:  icmp_seq=11 ttl=64 time=11.161 ms
64 bytes from 10.0.2.1:  icmp_seq=12 ttl=64 time=11.002 ms
64 bytes from 10.0.2.1:  icmp_seq=13 ttl=64 time=11.297 ms
64 bytes from 10.0.2.1:  icmp_seq=14 ttl=64 time=11.023 ms
64 bytes from 10.0.2.1:  icmp_seq=15 ttl=64 time=10.641 ms
64 bytes from 10.0.2.1:  icmp_seq=16 ttl=64 time=11.359 ms
64 bytes from 10.0.2.1:  icmp_seq=17 ttl=64 time=11.299 ms
64 bytes from 10.0.2.1:  icmp_seq=18 ttl=64 time=10.914 ms
64 bytes from 10.0.2.1:  icmp_seq=19 ttl=64 time=11.429 ms

--- 10.0.2.1 ping statistics ---
20 packets transmitted, 14 packets received, 30.0% packet loss
round-trip min/avg/max/stddev = 10.641/11.225/11.956/0.337 ms
```

As a result of Dummynet intercepting the traffic, packets 3,4,5,7,9 and 10 are dropped as they are sent, a total of 30% as specified. It should be noted that as packet losses in Dummynet are probabilistic, another ping may result in different packets and even a different number of packets being dropped.

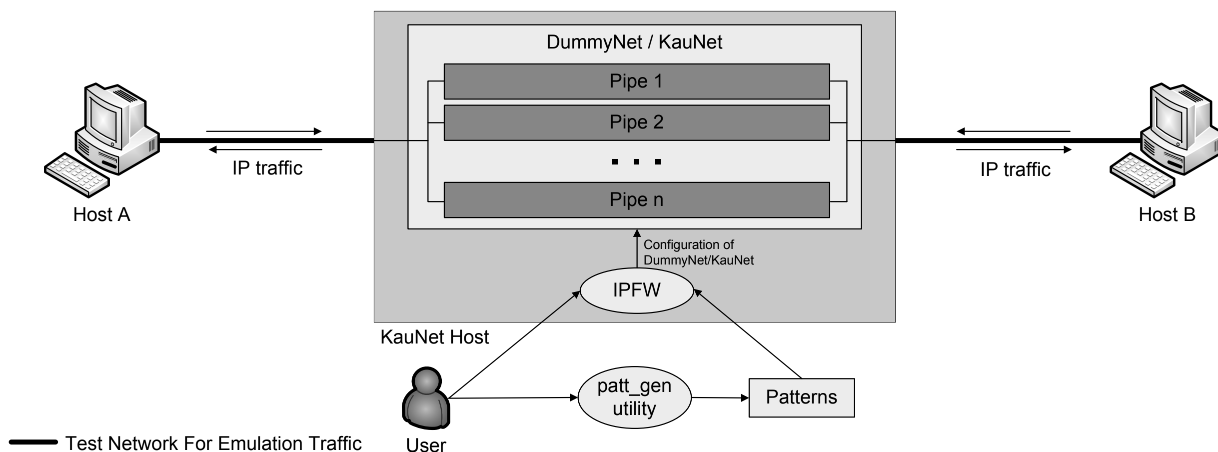

Figure 2.6: KauNet - Emulation overview.

2.3 KauNet

Two problems faced when working with Dummynet is the difficulty in reproducing tests and the difficulty in controlling the emulated environment. The former is primarily a result of the use of probabilistic emulation effects, while the latter comes from the use of a command line tool (IPFW) for configuration of the environment, requiring changes to the emulation to be made manually. Reproducing an experiment would therefore require very exact synchronization of Dummynet, scripts and traffic generation tools. Using probabilistic emulation effects, accurate reproduction may not be possible at all. KauNet [12] is an extension to Dummynet attempting to alleviate these problems by introducing the ability to emulate deterministic emulation effects.

Its functionality include deterministic emulation of the same features that Dummynet can emulate; bandwidth limitations, packet losses and delays, in addition to the probabilistic emulation of the original Dummynet[1]. The important difference is that unlike the original Dummynet, KauNet emulation effects are deterministic and controlled using premade patterns. Patterns are data files generated using the included pattern generation utility (*patt_gen*, see Section 2.3.2), either manually by a user or from traffic logs or other

[1]KauNet also introduces a new emulation effect; the ability to apply bit-errors to packets.

real world data sources. The patterns support both a data- and time-driven mode, acting on a per packet basis (in data-driven mode) or per millisecond basis (in time-driven mode). Through the use of deterministic patterns, experiments have a great deal of control over the environment as well as offering a high degree of reproducibility. A usage example illustrating deterministic packet loss is included in Section 2.3.3.

Dummynet and KauNet are integrated with a patched IPFW for setup and configuration of the emulation environment. The patterns created using *patt_gen* are inserted into KauNet using a new configuration option in IPFW, allowing a user to load KauNet patterns for specific pipes, as illustrated in Figure 2.6.

2.3.1 Patterns

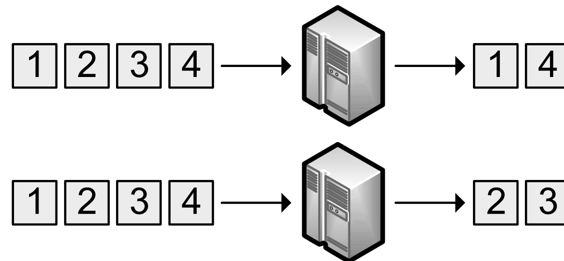

Figure 2.7: Example 1 - Dummynet with 50% packet loss.

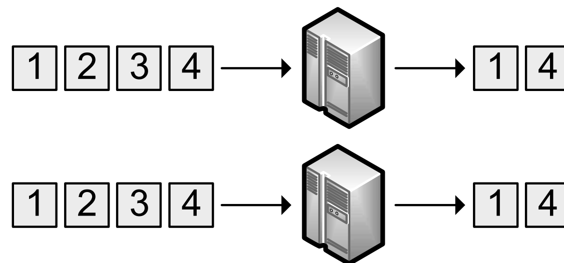

Figure 2.8: Example 2 - KauNet with 50% packet loss.

The use of patterns in KauNet is a deterministic method to describe the characteristics of a network. For example, Dummynet without KauNet can drop packets with a certain

Characteristic	Data-driven mode
Packet Loss	The specified packets are lost
Bandwidth Change	Changes take place after the specified number of sent packets
Delay Change	Changes take place after the specified number of sent packets
Bit-errors	Flip bits at the specified positions

Table 2.1: Pattern behavior in data-driven mode.

Characteristic	Time-driven mode
Packet Loss	All packets within the specified time periods are lost
Bandwidth Change	Changes take place at specified points in time
Delay Change	Changes take place at specified points in time
Bit-errors	Flip bits at the specified points in time

Table 2.2: Pattern behavior in time-driven mode.

probability, meaning that two test runs might drop different packets. In Figure 2.7, four packets are received by the Dummynet host and, in the first run, the second and third packet are dropped. In the second run, four packets are received by the Dummynet host and the first and fourth packet are dropped. With a KauNet pattern loaded however, test runs are always the same and the pattern also describes which packets to drop. In both test runs in the second example (see Figure 2.8), four packets are received and the second and third packet are dropped. As KauNet is using patterns to know which packets to drop, each new run will result in the same dropped packets.

All patterns may be configured to operate in one of two modes; data-driven mode or time-driven mode. Data-driven patterns use incoming packets to step forward in the pattern files, while time-driven patterns step forward each millisecond (an exception being bit-error patterns, see below). Patterns can control four different characteristics; packet loss, bandwidth changes, delay changes and bit-errors. A combination of the two modes and the four characteristics produce different behaviors for the network (see Table 2.1 and Table 2.2).

Each loaded pattern has an index to its current position in the pattern file. Bit-error patterns in both data-driven mode and time-driven mode are coupled with the bandwidth

restriction in the pipe, where the bandwidth specifies the number of bits to advance in the pattern file. In time-driven mode, the index is incremented every millisecond, even if no packets have been received, while data-driven pattern need a received packet to advance. For the other characteristics (packet loss, bandwidth change and delay change) the index is incremented one step for every millisecond or for every received packet, time-driven mode or data-driven mode respectively. The default behavior when reaching the end of a pattern is to wrap-around and start over. It is also possible to add an additional pattern to continue emulation using the second pattern after the end of the first pattern [19].

An illustration of an example with packet losses in the different modes can be seen in Figure 2.9 and Figure 2.10. The created pattern is of length four and set to drop the second and fourth packet (0101, a zero to forward the packet successfully and a one to drop the packet). As seen in the picture (Figure 2.9), the first, third and fifth (the first packet in the second run) packet are forwarded successfully, while the second and fourth packet are dropped. The pattern is wrapped around itself, meaning the sixth packet received would be dropped and the seventh packet would be forwarded successfully, and so on.

The same pattern, as used in the previous example, is applied in the next example. As this is a time-driven pattern, the pattern indicates that all packets received in the second and fourth millisecond would be dropped. This is also illustrated in Figure 2.10. In this example, two packets are received each millisecond, they are either dropped (a one) or forwarded successfully (a zero).

For KauNet to be able to use a pattern, it need to be inserted into the kernel space ahead-of-time. This is done by using the IPFW command.

```
ipfw pipe 100 config bw 1Mbit/s delay 10ms pattern created_pattern.pat
```

This command configures an already existing pipe (ID 100) and sets the bandwidth to 1 Mbit/s, the delay to 10 milliseconds and to use the created pattern in *created_pattern.pat*. The pattern is inserted into kernel space in a compressed format, where it is controlled by KauNet.

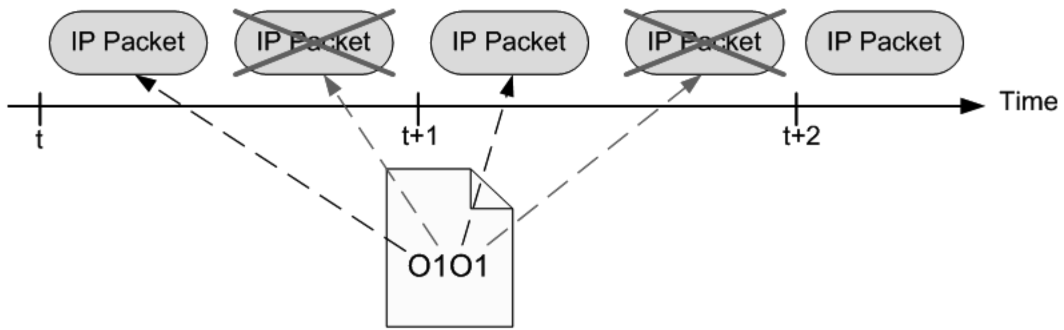

Figure 2.9: Data-driven packet loss example [19].

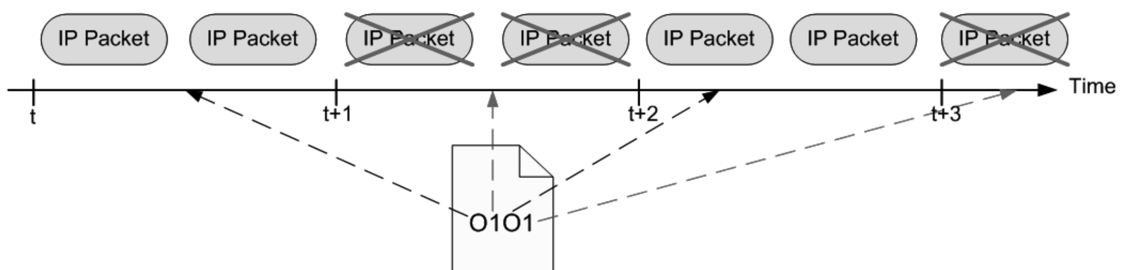

Figure 2.10: Time-driven packet loss example [19].

In addition to regular patterns, KauNet supports grouping patterns together into scenarios. Scenarios are, like patterns, created with *patt_gen*, but are essentially simply collections of patterns.

2.3.2 Pattern Generation Utility

patt_gen is a command line pattern generation tool which can both create and manage patterns. Pattern content can be entered on the command line directly or imported from simple text files containing uncompressed pattern descriptions. The pattern descriptions can be created from different kinds of sources, for example, simulator results, traffic logs or hand crafted.

```
patt_gen <type> <how to generate> <filename> <mode> <size> <positions>
```

There are four types of patterns available for the pattern generation; packet loss, bandwidth change, delay change and bit-error. Each of these types can be generated in a pseudo-random way, using the Gilbert-Elliot model [20][21] or by specifying positions or intervals directly (or in a file). The size parameter has a different unit depending on the selected mode. Packet loss, bandwidth change and delay change use the unit *packets* in data-driven mode and the unit *milliseconds* in time-driven mode. The fourth characteristic, bit-error, uses the unit *kilobyte* in both data-driven and time-driven mode. Due to *patt_gen* compressing the pattern file, there is only an indirect connection between size and disk space. The position parameter describes all positions (or intervals) used in the pattern.

For convenience, there is also a graphical user interface (GUI) available, called Pattern generation GUI (*pg_gui*).

2.3.3 KauNet Usage Example

This usage example is illustrated in Figure 2.6. Host A (10.0.1.1) sends ping packets to Host B (10.0.2.1) via a KauNet emulator.

The following is an example of a packet loss pattern used in KauNet.

```
KauNet# ./patt_gen -pkt -pos plp.pat data 20 5,6,8,12,14,17
```

patt_gen is in this example used to create a packet loss pattern by hand. The first parameter tells *patt_gen* that this is a packet loss pattern. The second says that this pattern is position-based, i.e. the pattern is triggered for specific packets or milliseconds. The third is the file the pattern will be written to. The fourth says that this pattern will be data-driven, i.e. advanced for each packet. The fifth defines the length of the pattern, in this case that it will cover 20 packets before wrapping around. Finally, the series of numbers defines the specific packets to be dropped.

```
KauNet# ipfw add 1 pipe 42 icmp from 10.0.1.1 to 10.0.2.1 out

KauNet# ipfw pipe 42 config bw 10Mbit/s delay 10ms pattern plp.pat
```

IPFW is now used to create a rule matching any outgoing ICMP traffic (from 10.0.1.1 to 10.0.2.1), along with a pipe to filter this traffic through. The pipe is configured with a fixed bandwidth and delay, and the KauNet pattern file is loaded.

```
Host A# ping -i 0.1 -c 20 10.0.2.1
```

Using ping, 20 ICMP packets are sent to a network connected machine.

```
PING 10.0.2.1 (10.0.2.1): 56 data bytes
64 bytes from 10.0.2.1: icmp_seq=0 ttl=64 time=11.545 ms
64 bytes from 10.0.2.1: icmp_seq=1 ttl=64 time=11.039 ms
64 bytes from 10.0.2.1: icmp_seq=2 ttl=64 time=10.983 ms
```

```
64 bytes from 10.0.2.1: icmp_seq=3 ttl=64 time=11.017 ms
64 bytes from 10.0.2.1: icmp_seq=6 ttl=64 time=11.548 ms
64 bytes from 10.0.2.1: icmp_seq=8 ttl=64 time=11.074 ms
64 bytes from 10.0.2.1: icmp_seq=9 ttl=64 time=11.253 ms
64 bytes from 10.0.2.1: icmp_seq=10 ttl=64 time=11.430 ms
64 bytes from 10.0.2.1: icmp_seq=12 ttl=64 time=11.009 ms
64 bytes from 10.0.2.1: icmp_seq=14 ttl=64 time=11.115 ms
64 bytes from 10.0.2.1: icmp_seq=15 ttl=64 time=11.269 ms
64 bytes from 10.0.2.1: icmp_seq=17 ttl=64 time=11.307 ms
64 bytes from 10.0.2.1: icmp_seq=18 ttl=64 time=11.758 ms
64 bytes from 10.0.2.1: icmp_seq=19 ttl=64 time=11.679 ms

--- 10.0.2.1 ping statistics ---
20 packets transmitted, 14 packets received, 30.0% packet loss
round-trip min/avg/max/stddev = 10.983/11.288/11.758/0.255 ms
```

The results show that packets with sequence numbers 4, 5, 7, 11, 13 and 16 are dropped, corresponding to the positions defined by the pattern, for a total of 30% loss. Note that subsequent runs using the same pattern would result in the same packets being dropped as the packet loss pattern is deterministic. Compare this to the unmodified Dummynet example in Section 2.2.3, where another experiment using the same emulator configuration could result in vastly different results.

2.4 Summary

Evaluating performance and behavior of both new and old systems are an important part of developing new protocols and software for use in a network environment. Evaluation may be performed using theoretical analysis and models, the use of simulators and emulators to

create a fully or partially virtual test bed, or the use of real networks such as the Internet or a custom built test bed. Each method offers both advantages and disadvantages and are often complementary and beneficial for certain scenarios.

The focus of this thesis is emulation, specifically emulation using Dummynet and its extension KauNet. Dummynet is a well known emulator originally implemented on FreeBSD, a UNIX-like open source operating system, and has since been ported to several other platforms. It offers several emulation options including bandwidth restrictions and delays, as well as probabilistic packet losses over multiple links and packet classification using the IPFW application. KauNet extends Dummynet by introducing deterministic emulation of the previously mentioned options on top of the original Dummynet functionality. This is done through the use of patterns, separate data files generated by hand or real world data sources loaded into the emulator. KauNet thus offers greater control over the emulated environment and a higher degree of reproducibility.

Chapter 3

Problem Description

This chapter provides an overview of KauNet and its capabilities, as well as an introduction and motivation for the new functionality that is to be implemented in this thesis; the trigger pattern and trigger forwarding. Triggers are a method of passing information from KauNet to external applications and may be used for a variety of purposes, such as simulating cross-layer information or providing real-time emulation information. The chapter describes the desired functionality and the considerations that had to be taken into account when designing and implementing the trigger pattern and forwarding mechanism.

3.1 Background

KauNet is under development at Karlstad University and is used in research both at the university and at international research institutions such as ISAE in Toulouse, France. KauNet is a network emulation system and emulates deterministic end-to-end network characteristics such as bandwidth changes, delay changes, packet losses and bit-errors. The deterministic behavior is controlled by patterns, which can be based on many different sources of input, for example simulation results, actual measurements, statistical properties, or be hand-crafted. The patterns are run in two different modes; time-driven

and data-driven mode. In time-driven mode, the pattern is advanced one step each millisecond, while in data-driven mode, the pattern is advanced for every received packet. One exception is bit-errors, which operates in data-driven mode and are more fine-grained (advances on per bit basis).

The next section of this chapter will introduce a number of examples where the use of triggers can provide additional functionality to the emulator.

3.2 Motivating Examples

Triggers can be used for a variety of purposes. A very simple usage scenario would be to send emulation statistics up from KauNet to allow detailed real-time logging of the emulation, as opposed to the on-demand statistics available via IPFW. Evaluation of algorithms estimating link properties such as delay or available bandwidth may benefit from access to the actual link properties to compare to the estimation. Another possibility is to use it for real-time updating of the current emulation status, which may be useful for both statistics gathering and synchronization. However, triggers are not limited to distributing existing statistics, but can also be used to support additional functionality in the emulation, such as emulating cross-layer information at the end hosts for usage in evaluation of applications and protocols utilizing cross-layer optimization.

3.2.1 Real-Time Emulation Updates

Modification to the experiment setup during the course of the experiment requires careful synchronization of all components involved. For instance, a scenario may involve changing the emulator configuration to add a new link or remove an existing one at a specific time, or to start or stop a traffic generation tool at specific points during the emulation to simulate a network event taking place. This may be both difficult and time consuming to solve using scripts on multiple machines. By using trigger patterns, this synchronization can be

done easily using the existing setup and triggers. It will then become a simple matter of creating a trigger pattern that raises events at the same time as a different loaded pattern. Instead of trying to estimate when an internal emulator event takes place, the emulator itself can then use triggers to inform other applications of the event with a high degree of accuracy, and they in turn can react accordingly. Another more general use case for this type of trigger usage is logging.

3.2.2 Cross-layer Optimization

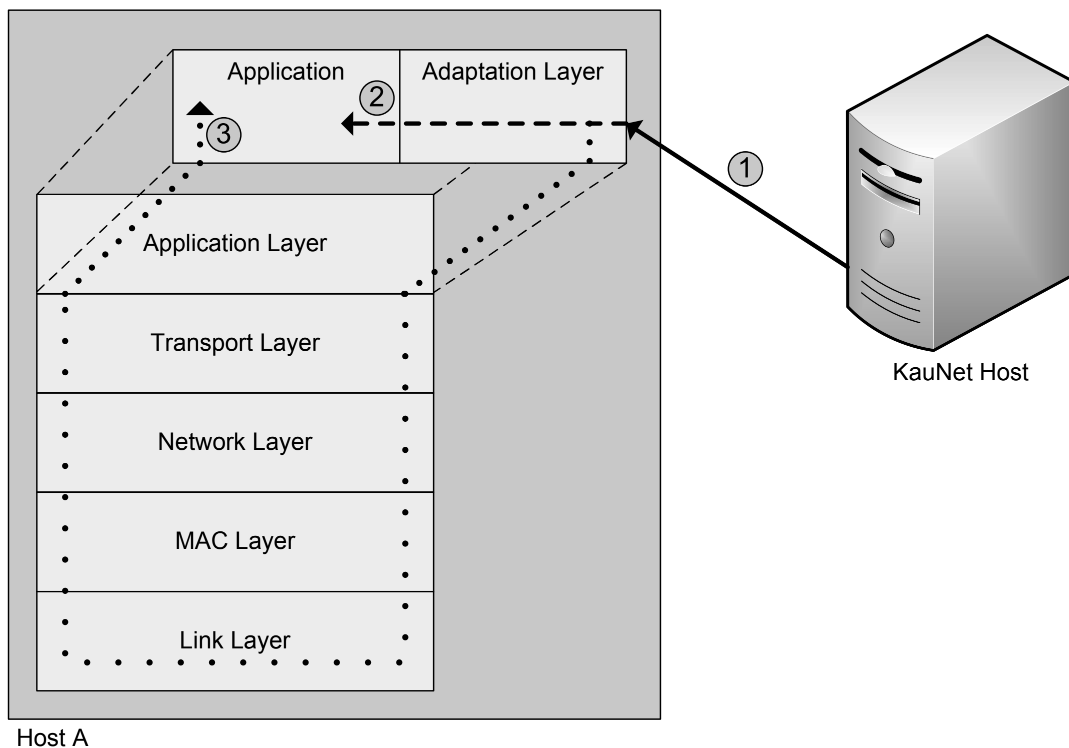

Figure 3.1: KauNet Trigger Example.

A candidate for the use of trigger patterns is the cross-layer optimization scenario. In a real network, an application can receive and react to information, such as the signal strength for a wireless connection, from interfaces residing in the lower layers in the network stack. For an emulated link, this is not possible as the link will actually be a wired connection with

relatively simple characteristics attempting to emulate complex characteristics. In these cases, it would be beneficial for the host to be able to receive information from the various layers of the network stack. A simple example is the estimation of available bandwidth using link information, such as the aforementioned signal strength. Another example is wireless mobile ad-hoc networks (MANETs). As the topology in these networks changes, links may go down unexpectedly and cause packet losses until a new route is established. As a result, they often show significantly degraded TCP performance (cf. [22][23][24]).

The loss recovery service provided by TCP assumes that any packet loss is caused by excessive traffic on the link, engaging the TCP congestion control mechanism [25]. The congestion control will attempt to prevent further losses by reducing the transmission rate (in implementations such as New Reno [26], the rate is reduced by half). However, if a congested network is not the cause, reducing the throughput will not solve the actual problem. Furthermore, once a new route is established, the throughput may take some time reaching its previous level (cf. the congestion avoidance algorithm [25]). By exploiting cross-layer information, TCP can be made aware of the cause of the packet loss and react more appropriately. While insufficient bandwidth may still be dealt with using the congestion control, in case of a routing failure, it may be more efficient to resume transmitting at the former rate once a new route has been established [27]. As a final example, TCP Quick-Start [28] makes use of cross-layer information for initial bandwidth estimation. As a connection is established, the sender will send a network layer request for bandwidth along the routed path. Intermediate routers may either accept, deny or negotiate a different bandwidth.

Using triggers and a suitable receiving application, these types of cross-layer information can be emulated and their effects evaluated, as illustrated in Figure 3.1. In this setup, host A is connected to a KauNet host used to emulate a wireless link. Triggers are used here by KauNet to send a notification to a subscribing adaptation layer on host A (1), which will receive and interpret the event as emulated cross-layer information (e.g. signal strength). The adaptation layer can then transform and forward the event directly to the application

(2), while the application will interpret the information as cross-layer information from the network stack (3). This allows events and information specific for routers or wireless networks to be emulated even when using a wired test bed without any of the actual hardware.

3.2.3 Link Properties Estimation

Many applications and protocols rely on algorithms to estimate network resources or properties, such as delay or available bandwidth [29]. In TCP, these types of estimations can be used to set appropriate protocol parameters based on the expected delay or adjust the congestion control mechanism to allow a higher throughput based on the expected bandwidth. An example is the Westwood TCP implementation (TCPW) [30], which uses bandwidth estimation to control the sending rate. Instead of halving the window and slowly building the rate back up using the congestion avoidance (as is the case in TCP implementations such as New Reno [26]), TCPW resets the window size to a value based on the estimated bandwidth [31]. This means that Westwood is less sensitive to random losses than New Reno by not overreacting by assuming congestion and cutting the transmission rate immediately [32].

When testing the accuracy of these algorithms, triggers in KauNet offer a method of sending the actual values that is used in the emulation directly to an application where the estimated value can be compared to the real value.

3.3 Design Considerations

In the last example of the previous section, KauNet triggers are used to report the current bandwidth on the emulated link to an application. Any changes to the bandwidth should therefore be reported. As modifications to the emulated environment in KauNet are caused by patterns being invoked, including the emulated bandwidth, a trigger pattern that can

run synchronized with the other types of patterns is therefore desired. To accomplish the synchronization, trigger patterns should have the same structure and granularity as the other KauNet patterns.

For an adaptation layer to receive triggers generated in KauNet, the KauNet host need a mechanism to forward the triggers. The adaptation layer can reside on either the local machine (the KauNet host) or on a remote machine. Different methods should be evaluated for making a decision about how to pass information to the adaptation layer. The threshold for the trigger passing rate should be at least one event per millisecond, corresponding to the resolution of time-driven KauNet patterns.

3.4 Summary

While KauNet supports on-demand access to emulation statistics, it is desirable for a mechanism to allow immediate notification of events. Event information can be used for a variety of purposes, such as emulating cross-layer information not otherwise available or provide real-time statistics and emulation state updates. To facilitate this, an additional pattern, the trigger pattern, is to be implemented to send events with arbitrary data from KauNet to an adaptation layer on a local or remote machine. As with the regular KauNet patterns, the trigger pattern should support both data- and time-driven modes and be created using the pattern generation utility. In order for events to be forwarded, the implementation of an appropriate inter-process communication mechanism is necessary. Various methods are available, and to ensure a working design that can satisfy the demands, these should first be evaluated in terms of, primarily, reliability, maximum event frequency and maximum bandwidth.

Chapter 4

Design

This chapter introduces and describes the general design and use of the trigger functionality to be added to KauNet, as well as necessary modifications to the existing platform. The trigger functionality is composed of two parts; the trigger passing mechanism that sends information from KauNet, and the trigger pattern that controls it. This in turn requires the creation of new components for both sending and receiving events raised by the triggers. Described in this chapter are the design choices made for both the trigger patterns and the trigger passing mechanism. Motivating the design choices made is an experiment that evaluates different possible implementations of an inter-process communication mechanism, as well as a description of the new trigger passing components that are to be added.

4.1 Triggers

Triggers are internal events taking place within KauNet, and are used as a mechanism for passing information out from the FreeBSD kernel. As described in Section 3.2, triggers are useful for several purposes, such as emulating cross-layer information or automatic real-time emulation updates. KauNet triggers are governed by the trigger pattern that may be inserted into the emulator in the same fashion as any other KauNet pattern. The trigger

pattern is a value pattern, meaning it can store arbitrary values for each position in the pattern. These values are used as the trigger value that is passed when an event is raised. This enables a user to generate trigger patterns using predefined values that can be used to describe the different types of events taking place, as well as any related information that should be passed. As with the other types of KauNet patterns, the trigger pattern can use both time- and data-driven modes of operation. This means the trigger pattern can be run synchronous to any other pattern loaded in the emulator[1]. Therefore, it is possible to generate events at very precise points in time or at the arrival of specific packets, matching the occurrence of another pattern event. A trigger pattern could for example be generated based on an existing pattern to raise events whenever a packet is dropped or when the emulated bandwidth exceeds a specific threshold. The trigger value could then be used to describe the type of event and/or the sequence number of the dropped packet or the exact bandwidth.

4.2 Trigger Passing

Trigger patterns add the ability for KauNet to generate events internally, but for this to be useful in the general case, some sort of mechanism allowing observers outside the KauNet kernel to be notified of these events is necessary. The purpose of the trigger passing mechanism is to allow internal KauNet triggered events to be forwarded to subscribing adaptation layers (see Section 4.2.4). Ultimately, the trigger passing mechanism should be able to pass any event raised to an arbitrary number of adaptation layers running on both local and remote machines. As neither KauNet nor the trigger pattern is aware of the semantic meaning of the trigger value, the interpretation of the value is left to the receiver. Therefore, it will also be necessary for the trigger passing mechanism to define the structure of the data so that it can be accessed and parsed correctly.

[1] An exception is the bit-error pattern that uses a higher resolution than the per packet basis advancement of the other data-driven patterns.

The chosen mechanism must be able to pass a minimum of one event per millisecond up from the kernel (the maximum event frequency for a time-driven trigger pattern), preferably out to a local and remote receiver directly and with a margin of error. It should also be able to pass triggers without any losses, assuming any intermediate network does not cause packet losses. Finally, it should be able to pass the trigger in a timely manner so that it is not outdated by the time it arrives at the receiver, barring any network delays beyond the control of the mechanism. Furthermore, it should preferably be as easy to implement, modify and use as possible.

4.2.1 KauNet

In order to avoid unnecessary modifications to the KauNet kernel, it was decided that KauNet should not contain the implementation of the trigger passing functionality (thus keeping the kernel code cleaner and avoiding the need to frequently recompile the kernel during development). Instead, this is implemented in a separate kernel module that can be loaded and unloaded independently. The module itself provides an interface for KauNet to use when it wishes to pass a trigger. Because of this, only a few lines of code need to be added to KauNet to support the trigger passing mechanism, and changes to the trigger passing implementation should require no or only a few changes to KauNet itself.

4.2.2 Trigger Passing Mechanism Evaluation

In the design of the KauNet Communication Module (see Section 4.2.3), consideration had to be taken to ensure that triggers and the associated data could be handled at a sufficient rate. As events are potentially triggered continuously, it must be capable of passing the events in a timely manner. The design chosen for the communication module must be able to satisfy this demand, but in addition to performance and reliability, other factors such as complexity, scalability and adaptability/expandability should also be evaluated. Performance and reliability are evaluated as the maximum rate the mechanisms are able to

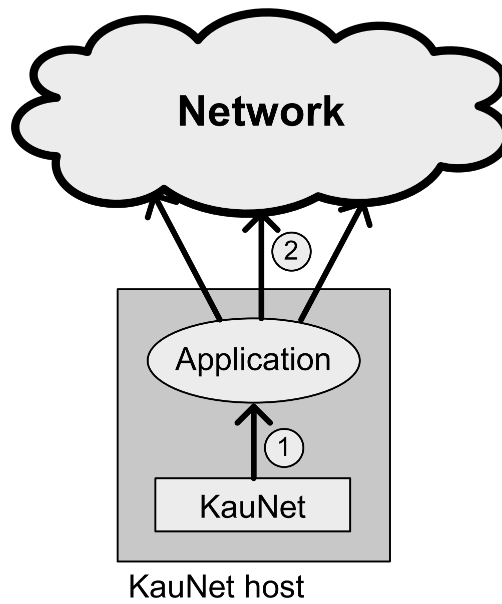

Figure 4.1: Passing triggers using an IPC mechanism.

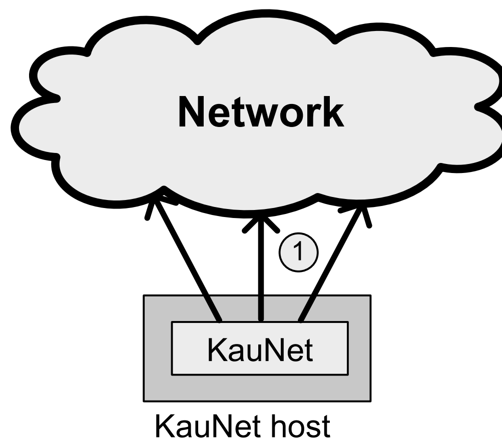

Figure 4.2: Passing triggers using network sockets.

send information without losses. The remaining factors are not evaluated experimentally, but subjectively valued based on how easy a system using the different mechanisms is to design and implement, and modify in case higher rates or payloads are required. For this reason, tests were performed using different methods to pass triggers from the kernel up to the userspace. Two methods were chosen for evaluation; using an inter-process communication (IPC) mechanism, or using sockets. An IPC mechanism would require two separate stages; first passing events up from the kernel to an intermediate application (1) and then sockets to forward them out (2), as illustrated in Figure 4.1. Using sockets would allow the first step to be bypassed entirely by forwarding the events directly from the kernel (see Figure 4.2). For the first method, the IPC mechanisms evaluated were POSIX [33] signals and UNIX domain sockets. For the second, network sockets using the UDP [34] protocol was tested.

Without an existing trigger pattern, a way to simulate events being sent from KauNet to an external application had to be implemented. In order for the test to be as accurate and relevant as possible, the events should be generated in the kernel and forwarded to a userspace application. Rather than implementing the experiment code directly into the KauNet code, a kernel module for generating events was created to avoid unnecessary contamination and repeated compilation of the kernel as well as offer easy modifications, while still allowing the experiment to take place in kernelspace. A simple userspace client application was implemented to receive the generated events.

Tests were performed to find the maximum reliable rate at different payload sizes for the various methods. The maximum reliable rate for each method and payload size was determined based on ten tests, running for ten seconds each. The number of losses (events generated and sent but not received) and errors (events not sent or sent incorrectly) were recorded, and the threshold for the maximum reliable rate was zero losses and zero errors. The total number of different tested payload sizes were 18, ranging from 4 bytes up to 1000 bytes. From the results of these tests, the maximum reliable bandwidth for each method

could be inferred.

Events were generated in bursts, followed by a pause of one millisecond (the shortest possible reliable pause for the kernel), in order to simulate a number of events taking place simultaneously within KauNet. This design was chosen in order to stress test the frequency at which events could be generated reliably, which is important for trigger patterns in data driven mode. As a result, the tests had to be timed to ensure that they could be performed in the allocated time. The tests that failed to do so were omitted from the results. An alternative using a continuous loop and timestamps instead of a pause in order to eliminate this problem was explored, but was deemed infeasible as the increased CPU load from the kernel module caused the tests to fail.

Two machines were used during the evaluation. The first machine (see Appendix Table A.1 for specification) was used in all tests, while the second machine (see Appendix Table A.2 for specification) was only used during the network socket test (remote). Even though the two machines have a Gigabit Ethernet card each (connected via a router), the tests were performed with the cards configured for 100 Mbit/s.

Evaluation of Signals

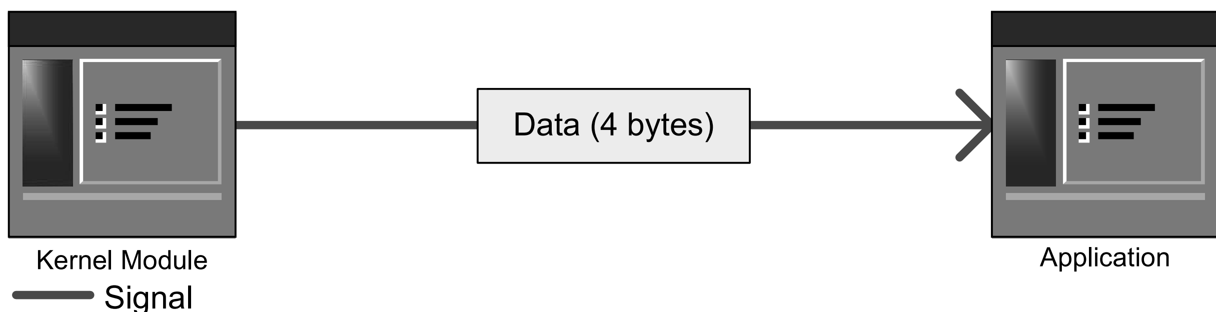

Figure 4.3: Raw signal data transfer.

A POSIX signal is an asynchronous IPC mechanism that may be used to notify a process of external events. Signals may be sent at any time from any process, the kernel or due to a hardware exception, to any process with a known process identifier. When a signal is

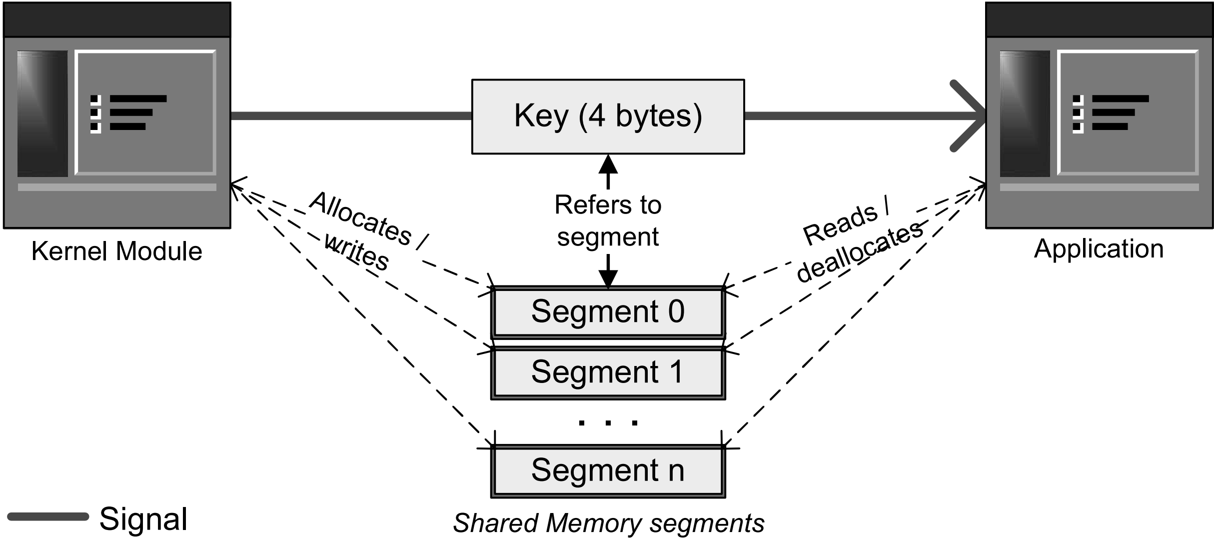

Figure 4.4: Dynamic shared memory data transfer.

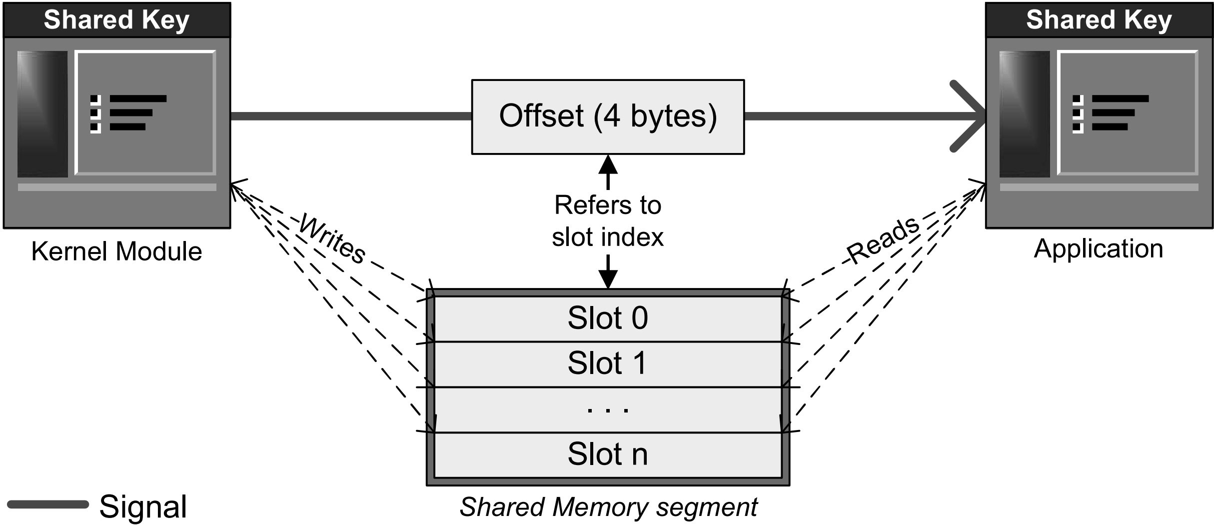

Figure 4.5: Static shared memory data transfer.

received, the operating system will cause an interrupt in the receiving process, suspending the normal flow of execution until the signal has been processed. Various default signal handlers are implemented that describes how a process should react to the signals, but a process may also register its own signal handlers to provide a custom functionality for the different signals (notable exception being the SIGKILL and SIGSTOP signals that may not be redefined).

In the experiments, the *sigqueue()* function was used to generate SIGUSR1 interrupts, allowing an integer (4 bytes of data) to be sent along with the signal. Two different approaches were examined; raw signals and signals with shared memory. Raw signals uses the attached data to transmit the event information directly (see Figure 4.3), while signals with shared memory uses the data as key (see Figure 4.4) or offset (see Figure 4.5) to a shared memory segment containing the event information. While raw signals are limited to 4 bytes of data per signal, the addition of shared memory allows for significantly larger payloads.

When using shared memory, two methods may be used to transmit the event information. The first is to use the signal data to include a key to a shared memory segment. The segment is created when the signal is generated and then deallocated at the receiver once the information has been read. This method is referred to as *dynamic shared memory* (DSM), as the segments are allocated and deallocated as needed. The second method creates a large shared memory segment at the start of the test, using a predefined key known to both sender and receiver. The signal data sent is then an offset to one of several slot in the shared memory segment, containing the event information. The information is written to the slot when the signal is generated and read once received. Once read, the slot can safely be reused by overwriting the content with new event information. In the tests, the memory segment contained 1000 slots to store information. This method is referred to as *static shared memory* (SSM), as shared memory is only allocated once and reused throughout the test.

In the case of shared memory, the event information is copied to the local memory on reception, but is otherwise ignored. This ensures that the shared memory is no longer needed and can safely be either deallocated or overwritten, while the information remains at the receiver to be processed when there is time. It also minimizes the risk of reaching the limit of the number of segments allocated at the same time, in the case of dynamic shared memory. It is important to note that when the receiver is interrupted by a signal, any signals received before the signal has been processed will be lost. Because of this, the processing time for the received information is relevant to the performance of signals as an IPC mechanism. As the processing becomes more time consuming, the more signals would risk being lost. To alleviate this problem, an alternative method was also tested. Rather than to let the interrupt handler take care of the processing, a separate work thread was used to process the event information instead. By using a separate thread for processing, the interrupt handler was left with the relatively simple task of storing the received key or offset for the work thread to process, ensuring that the receiver spent a minimum amount of time in interrupt mode. In these tests, the number of events that were not processed by the work thread by the time the test ended were also recorded. In the case of static shared memory, the number of times unprocessed memory was overwritten was also recorded, as this would essentially be equivalent to a loss.

Evaluation of Sockets

A socket is a bidirectional endpoint for data communications. Two sockets can send data back and forth by encapsulating it in a packet, containing the data itself and necessary headers. They may be configured as use in both local and remote communication, supporting a wide range of features, by using different domains and protocols. Network sockets (AF_INET family) uses Internet protocols such as UDP or TCP, IP addresses and ports for communication over networks. UNIX domain sockets (AF_UNIX family) specialize in local IPC communication, using file system path names for addressing.

Both network sockets and UNIX domain sockets were tested, the former using both a local client and one hosted on a network connected client for a total of three different test scenarios. For the network sockets, the tests were performed using the UDP protocol. While TCP does offer some desirable features, specifically retransmission of dropped packets in this case, it was decided that they would not be necessary for practical use. UDP was chosen instead to avoid the larger headers and greater overhead.

Unlike with signals, very few special considerations had to be taken into account. Socket communication feature a built-in buffer that automatically stores the packets until they can be processed. While signals had the possibility of events being overwritten or lost due to the receiver being busy, the buffer prevent such problems. Threaded processing was therefore not necessary. Problems arise only if the buffers are filled, in which case the arriving packets are dropped. The total number of packets not received were recorded, along with the number of errors caused by the buffer of the sender being filled.

Results & Conclusion

The maximum reliable rates for the tested methods are illustrated in Figure 4.6. Omitted from the graph are the test results for raw signals, as their payload can not be adjusted, remaining constant at 4 bytes per signal. For this payload, raw signals showed no losses up to 125 signals per millisecond. Signals using shared memory showed a correlation between the payload size and the maximum rate, more so for SSM than DSM. DSM signals maintained a rate around 45 signals per milliseconds for payloads up to 250 bytes, at which it dropped to 40 where it remained for the remaining payload sizes. Higher rates were tested and could be received, but the kernel was unable to generate the signals at an appropriate rate and these were therefore excluded. SSM signals by comparison allowed a significantly higher rate of about 100 signals per millisecond initially, but dropped sharply to 65 and then 60 at payloads of 450 and 800 bytes respectively.

UNIX domain sockets allowed a fairly high rate of 60 at the lower payloads tested, but

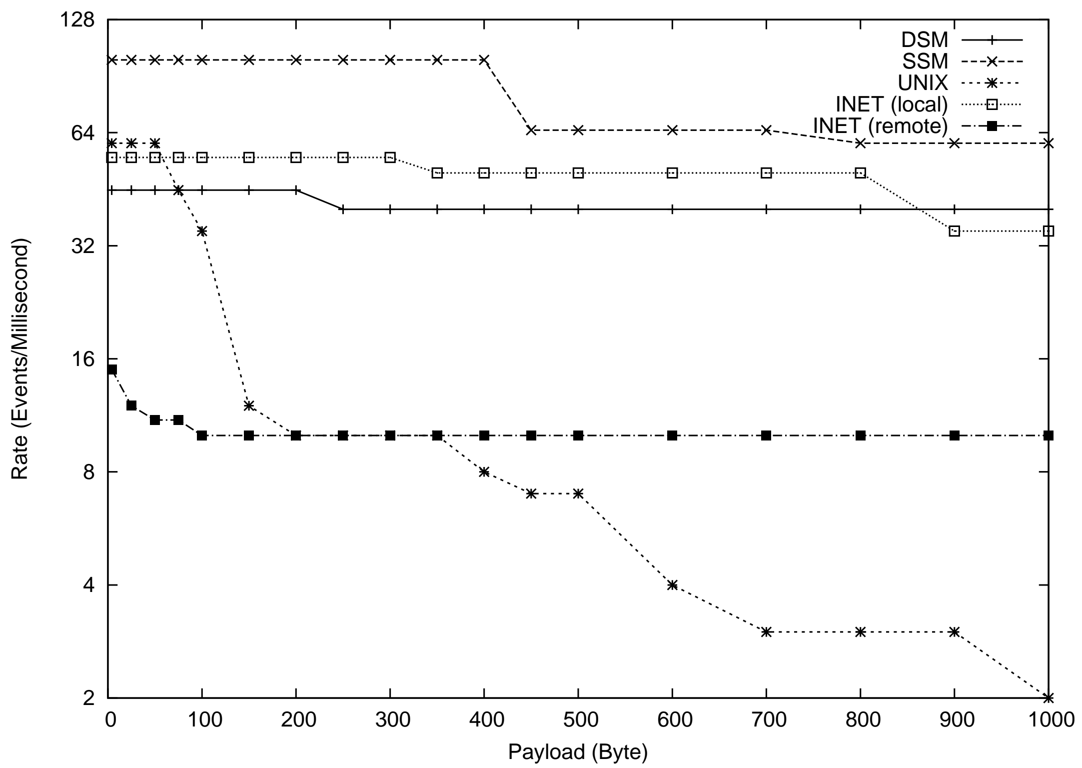

Figure 4.6: Maximum reliable rate as a function of the payload size.

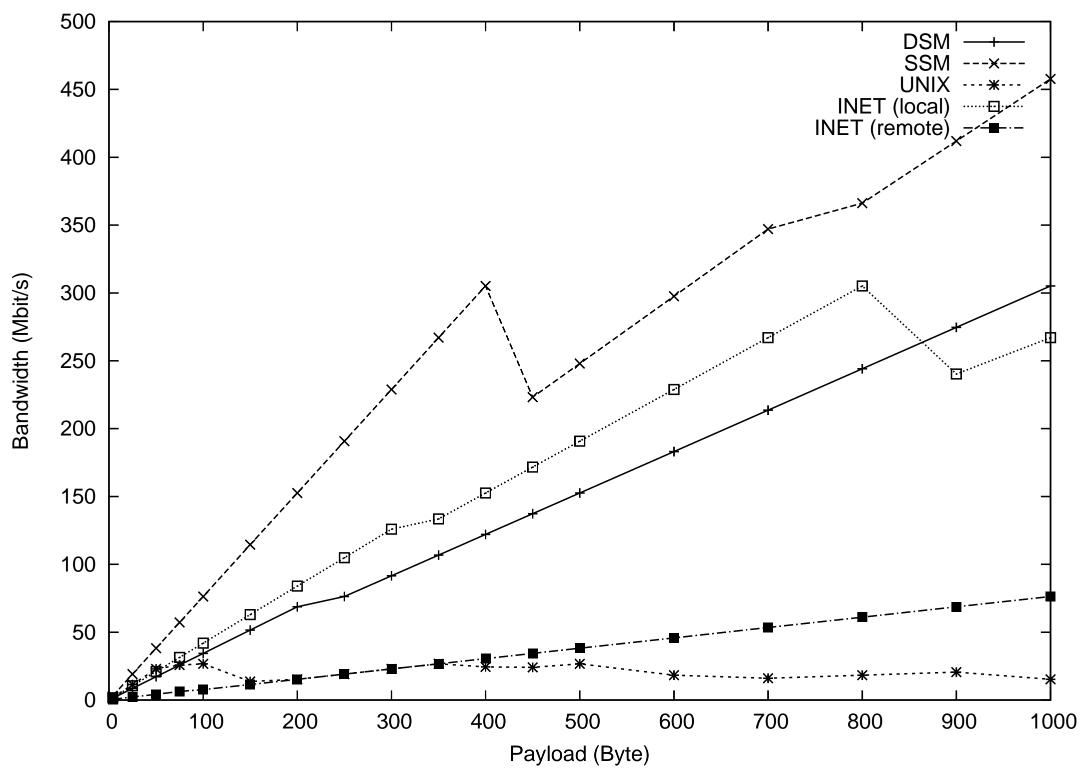

Figure 4.7: Maximum bandwidth as a function of the payload size.

quickly degraded with increased losses as the payload increased, dropping to as low as 2 per millisecond at a payload size of 1000 bytes. While all tests showed decreased performance if the system had other tasks to perform simultaneously, the UNIX domain sockets showed a high sensitivity to disk operations in particular, showing significant degradation.

Network sockets communicating with a local receiver showed initial rates similar to those of UNIX domain sockets at around 55 per milliseconds. The degradation was significantly lower however, dropping to 50 at a payload size of 400 bytes and then to 35 at 900 bytes.

Network sockets using a remote receiver initially had a rate of only 15 per millisecond, the lowest initial value of all the methods, which quickly dropped to 10 at payloads of only 100 bytes. However, it remained stable at this rate for the remaining payloads, outperforming or matching UNIX domain sockets at payloads equal to or greater than 200 bytes.

From these results, the maximum data bandwidth out from the kernel can be inferred. This is illustrated in Figure 4.7, with the bandwidth as a function of the payload size (using the maximum reliable rate for each size).
To summarize the results, the following can be noted:

- Raw signals are vulnerable to losses only at higher rates, but have a very limited payload that may make it undesirable for use.
- DSM signals are vulnerable to losses at medium and high rates but support larger payloads. Errors during sending may result in orphaned shared memory segments and thus memory leaks. There is also a maximum number of concurrently allocated segments, regardless of their size.
- SSM signals are vulnerable to losses at medium to high rates depending on the payload size, but allow the highest rates and bandwidth of all the tested methods. A long processing time may result in unread events being overwritten (effectively a

loss), but unlike the DSM method, does not result in a memory leak.

- UNIX domain sockets allow fairly high rates at smaller payloads, but quickly degrade as the payload increases. It also degrades significantly with simultaneous hard drive activity.

- Network sockets on a local machine allow rates similar to UNIX domain sockets, but significantly lower degradation with increased payloads.

- Network sockets and a remote receiver have the lowest rates and bandwidth of all the tested methods (although this may be caused by the network bandwidth rather than the socket mechanism itself). This method does however allow the event to be forwarded directly to other machines without an intermediate application to receive event from the kernel and then pass it on to external observers.

- Performance of the socket methods may be improved by using larger buffers for both the sender and the receiver.

- Overall performance and reliability may be improved by aggregating events using a larger payload and sending them further apart.

- Performance is likely to be significantly improved if using an evenly distributed event generation rather than bursts. The tests used to obtain these results are intended as a worst-case scenario.

It is important to note that these tests were done to evaluate the performance of the methods when sending data up to userspace from the kernel. Remote tests with network sockets are unique among these tests as they send data both up from the kernel and out to potential remote subscribers whereas the others are used for local communication only. While all methods will need to use sockets to send data out to subscribers eventually, they may do so using aggregation. The remote network socket tests were performed to evaluate

their performance as a trigger passing mechanism without aggregating events. Assuming aggregation of events, the remote network sockets results, especially for higher payloads, may also serve as an indication for the potential final data rate to remote subscriber for all methods.

Based on the evaluation, all the tested mechanisms are able to fulfill the minimum requirements in terms of performance and reliability with no aggregation required. The primary factors determining which mechanism to choose for implementation are therefore complexity of design and implementation, and how easily it can be modified for additional features and better performance. For these reasons, network sockets were selected, as motivated in the next section.

4.2.3 KauNet Communication Module

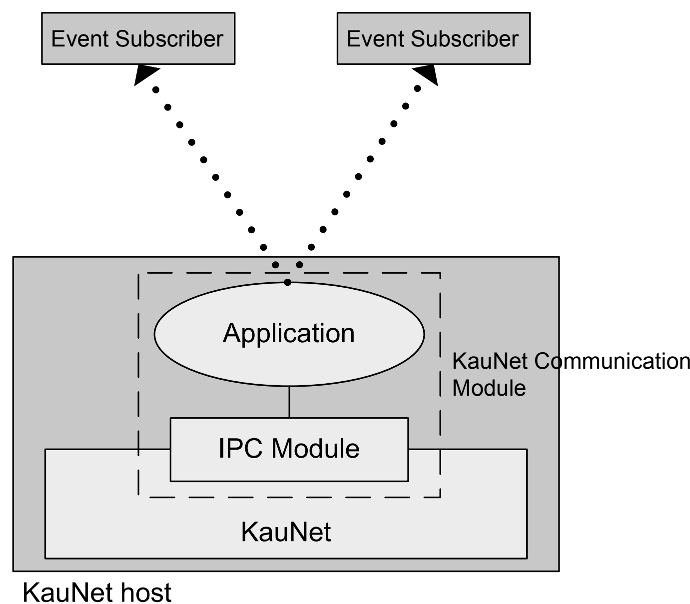

Figure 4.8: KauNet communication module using an intermediate receiver and IPC.

The trigger passing functionality of KauNet is provided entirely by the KauNet Communication Module (KCM). The KCM provides an Application Programming Interface (API)

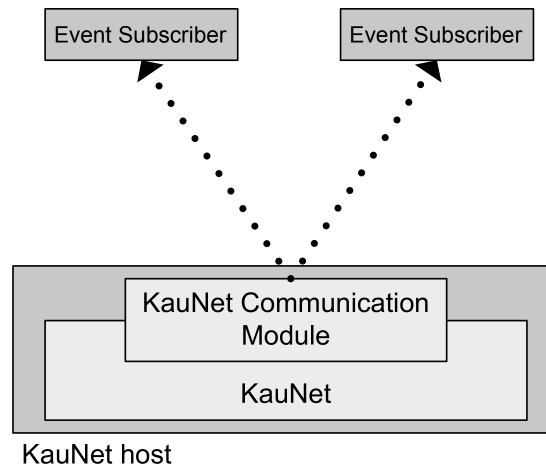

Figure 4.9: KauNet communication module using sockets.

for KauNet while remaining an independent module. KauNet uses an interface function in the KCM and simply passes the event data as arguments. The KCM is responsible for distributing the events received from KauNet to an arbitrary number of subscribers, both local and remote. To this end, it also acts as a server, accepting subscription requests from clients and keep track of subscribers. By implementing the KCM as a separate add-on module, it is possible to replace it with few or no changes required to KauNet as its internal workings are not relevant as long as its interface remains the same. It is even possible to load and unload different modules at any time during an experiment. Conversely, changes to KauNet should require few or no changes to the KCM.

The functionality of the communication module can be summarized as passing events up (from the kernel) and out (to local or remote subscribers). Two possible designs were considered; including a second component for receiving IPC communication from the kernel and then distribute it out using network sockets (see Figure 4.8), or implement the network sockets directly in the kernel and distribute the events to both local and remote subscribers directly (see Figure 4.9). If network sockets could not perform sufficiently, the former method could be used, using an IPC mechanism able to pass the information up from the kernel more efficiently, and the intermediate receiver could then forward the events

at its own pace, possibly aggregating them for better performance. The advantage of the latter method is instead that the design and implementation is simplified considerably by removing the need for the intermediate receiver, instead being able to use both local and remote communication directly.

Based on the results of the trigger passing mechanism evaluation (see Section 4.2.2), all methods are capable of providing the required functionality, although raw signals only support limited payloads (4 bytes). The second design using network sockets was therefore chosen. While the performance for data transfer out from the kernel is one of the lowest of the tested mechanisms, network sockets are still able to provide reliable transfer at a sufficient rates and payloads while also offering a simple implementation. Both aggregation and increased payloads can be implemented with very little difficulty to increase bandwidth and partially support a higher event rate. Should it still prove insufficient in certain scenarios (such as a dense data-driven trigger pattern that may generate events at a higher rate than the network socket can sustain), the overall design of the KCM ensures that it can be replaced by a different implementation with relative ease.

4.2.4 Adaptation Layer

The Adaptation Layer (AL) is an arbitrary application that acts as a client for receiving and processing KauNet events. It can run on either a local machine with KauNet, or a remote machine connected via a network. It interacts with KauNet via its communication module (ALCM, see Section 4.2.5), which provides the necessary interface functions for communicating with KauNet. Depending on the specific needs for an experiment, it is likely that a custom AL must be implemented.

4.2.5 Adaptation Layer Communication Module

The Adaptation Layer Communication Module (ALCM) is an API library for use by adaptation layers. Its purpose is to provide an interface to KauNet for use by the adaptation

layer, thereby allowing changes to be made to both KauNet and the adaptation layer while maintaining compatibility as long as it remains compliant with the API, as well as simplifying the implementation changes to existing adaptation layers or completely new adaptation layers. The ALCM interacts with the KCM, providing the adaptation layer with functions to connect to a KauNet machine and receiving events. It also provides functionality for parsing received data to extract specific information, such as the trigger value or any other information known to the ALCM (timestamps, emulator information etc.). This allows changes to be made to the structure of the data sent and received by the KCM and ALCM without any need to alter the adaptation layer. It also ensures backward compatibility, as long as information is not removed from the data.

4.3 Summary

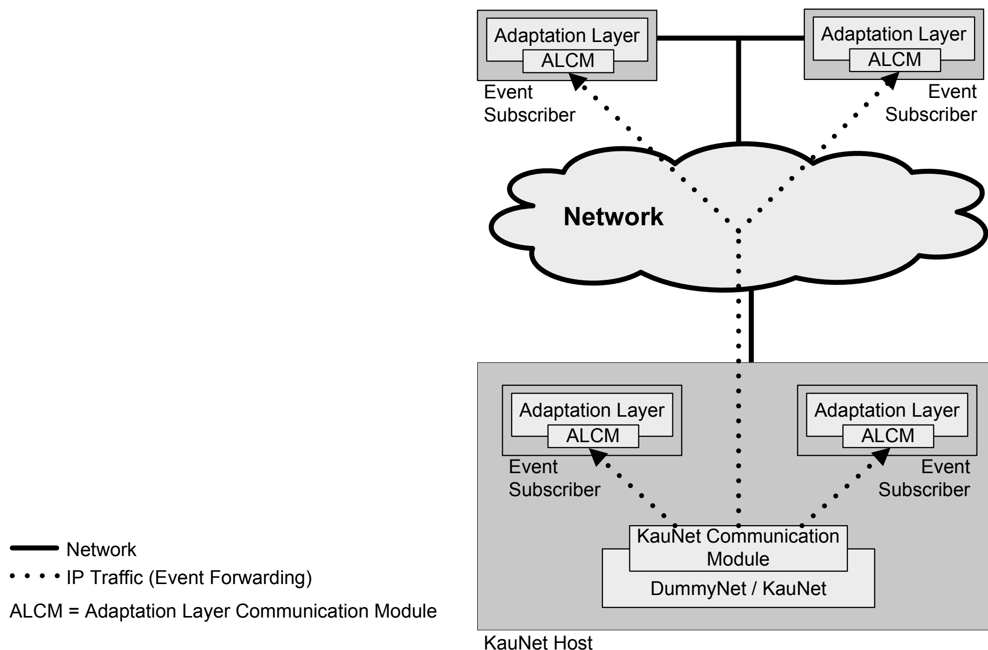

Figure 4.10: KauNet trigger passing.

Described in this chapter is the design of the communication modules for both KauNet

and the adaptation layer, a basic adaptation layer and trigger pattern for both time- and data-driven modes to allow trigger passing from KauNet (see Figure 4.10).

The purpose of the trigger pattern is to generate events at a specified point, that is passed up from the kernel, ultimately allowing the event to be forwarded to an adaptation layer on both the local and non-local machines. As with other patterns, trigger patterns are generated using *patt_gen*. They can be generated using existing patterns, in order to allow events to be generated when, for example, a packet is lost or when a bandwidth threshold is reached.

The KauNet Communication Module (KCM) is an add-on kernel module to KauNet. It is responsible for communication between KauNet and both local and non-local adaptation layers. To that end, it maintains a list of subscribers (adaptation layers that wish to receive KauNet events), handle subscription requests, and forward events to any subscribers. The Adaptation Layer Communication Module (ALCM) serves as the adaptation layer's counterpart to the KCM. It provides functionality for an adaptation layer to request subscriptions and receive events from KauNet, as well as parse the received information.

The adaptation layer is an application that receive and process events from KauNet. Depending on the application being tested, it can be implemented as a kernel- or userspace application. An arbitrary number of adaptation layers may run and subscribe simultaneously on both the local and non-local machines. In order to receive events, it must first send a subscription request to the KCM on the KauNet machine via the ALCM.

Chapter 5

Implementation

Described in this chapter are the implementation details of the trigger functionality in KauNet, composed of the trigger pattern and trigger passing mechanism. The chapter also includes an implementation overview of Dummynet and the inclusion of the KauNet patterns as a short background for the implementation of the trigger pattern processing. The implementation requires changes to be made to the existing KauNet code, but also to the pattern generation utility (*patt_gen*) to generate a new type of pattern, and the implementation of several new components that provides the trigger passing functionality. The chapter describes the structure, creation and processing of the trigger pattern itself, as well as the implementation in KauNet, the KauNet Communication Module, the Adaptation Layer Communication Layer API and an example Adaptation Layer.

5.1 Dummynet/KauNet Architecture

The Dummynet implementation is summarized in Figure 5.1. The *ready heap* stores a number of fixed-rate queues and handles scheduling based on bandwidth limitations. A low link utilization or high bandwidth would mean that packets are sent from the ready heap with little or no delay, while a congested link would result in more significant delays.

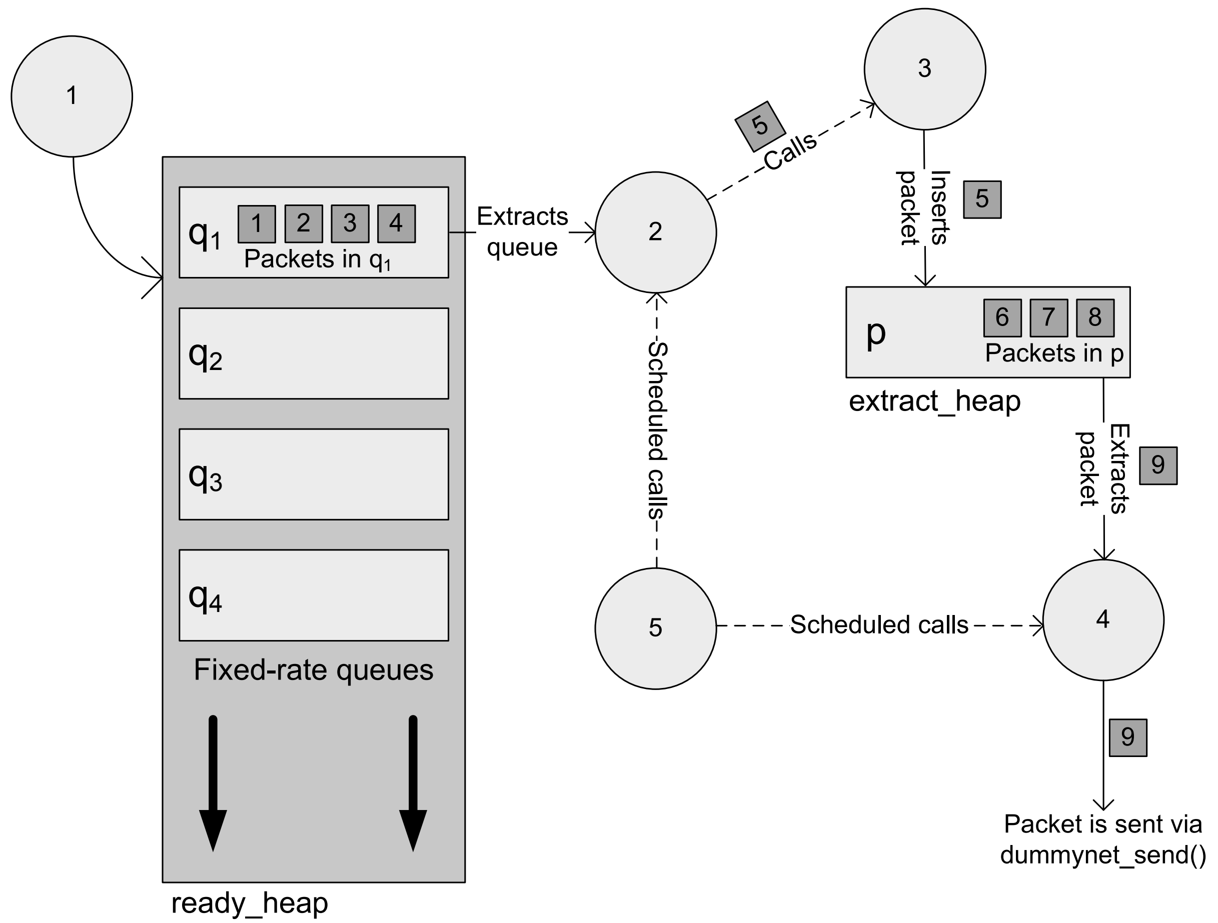

Figure 5.1: Dummynet event flow.

The *extract heap* (or delay line) contains packets that are scheduled to be sent based on fixed delays (simulating for example transmission and processing times for the packets). Marked in the figure are also the five primary Dummynet functions that are used for moving the packets internally across the emulated link, corresponding to the following:

1. Packets enter the emulator and are either dropped or queued. Bandwidth scheduling is applied for the *ready heap*.

2. Scheduler extracts queues from the *ready heap*, drains as many packets from them as possible based on available bandwidth, and scheduling for remaining packets in the queue is updated.

3. Packets are moved from the *ready heap* to the *extract heap* and fixed delay scheduling is applied.

4. The scheduler drains packets from the *extract heap* and prepares them for being sent out of the emulator back into the network stack.

5. The main Dummynet function executes once per millisecond, updates the internal clock and checks for scheduled events.

It is also in these functions that KauNet applies its pattern based emulation effects and advances its patterns. A more detailed description of the flow of events including the use of patterns is as follows:

1. Whenever a packet is about to be added to a queue in the emulator, *dummynet_io()* is called. If the random packet loss function of Dummynet or a KauNet packet loss pattern is present and determines that the incoming packet should be dropped, it will be dropped immediately without being moved further into the emulator. If it is not dropped, Dummynet will check the available bandwidth to see if it can be moved forward immediately and, if so, call *ready_event()*. If not, the packet is placed

in a queue, the expected output time of the packet based on the link bandwidth is calculated, and the queue is inserted into the *ready heap* to await its scheduled output time. In this function, the packet loss pattern for data-driven mode is advanced and both data- and time-driven packet loss patterns are invoked. In addition, data-driven bandwidth patterns are advanced, and any active bandwidth pattern is applied to the packet.

2. *ready_event()* is called whenever the scheduler is invoked to forward packets from a queue in the *ready heap* further into the emulator. The function first extracts the queue from the *ready heap*, then attempts to drain as many packets as possible from the queue, as long as there is sufficient bandwidth to do so. For each packet that is drained, the *move_pkt()* (3) function is called. If there are still packets left in the queue by the time the emulator has run out of available bandwidth, a new output time is calculated for the remaining packets and the queue is reinserted into the *ready heap* to await the next scheduling event. Finally, if the *extract heap* was empty prior to *ready_event()* being called, *transmit_event()* (4) is called. If a bandwidth pattern is active, it is applied here as the bandwidth scheduling is calculated.

3. *move_pkt()* is called by *ready_event()* whenever a packet is to be moved from the *ready heap* to the *extract heap*. The function calculates the output time of the packet based on any fixed delays specified by the pipe, then inserts the packet into the *extract heap* to await a scheduling event. KauNet uses this function to advance loaded data-driven delay patterns, and any active time- or data-driven delay pattern effects are applied here for the output time calculation.

4. *transmit_event()* is called when a packet is to be sent from the *extract heap*. The function will attempt to drain as many packets from the *extract heap* as is possible (i.e. all packets that are scheduled to be sent at this time). If a bit-error pattern is present and currently active, it is invoked in this function to modify the content of the

drained packet(s). The packets are eventually inserted into the network stack by the *dummynet_send()* function which is called periodically by Dummynet. Any packets remaining (i.e. those scheduled to be sent later) are reinserted into the *extract heap*.

5. *dummynet_task()* is the main function in Dummynet and is executed once for every tick of the kernel (Dummynet is designed for kernels running at 1000 Hz, meaning one tick per millisecond). The function updates the emulator's internal timer that is used by the scheduling mechanisms, as well as check for any events scheduled to take place. Depending on the event, *dummynet_task()* will call either *ready_event()* (for bandwidth based scheduling) or *transmit_event()* (for fixed delay scheduling) whenever necessary. KauNet advanced all its time-driven patterns here, as well as calculate the bits to flip (if any) for bit-error patterns.

5.2 Trigger Patterns

Trigger patterns control both when events are raised by KauNet and to an extent also what information these events contain. The implementation of the trigger pattern consists of two parts; defining the structure of the pattern, and a way in which they can be created. This section will describe in detail the content of the trigger pattern, as well as the syntax used for the pattern generation utility when generating new trigger patterns.

5.2.1 Pattern Structure

All KauNet patterns share a common structure, as does the new trigger pattern. Patterns are stored and imported into the kernel in a compressed format when loaded. The pattern structure is divided into seven parts; magic bytes, version number, pattern type, pattern size, data array size, indicator array, and data array. Each part is described below:

- Magic bytes: The numbers 75, 97, 117, 78, 101, and 116 are used to indicate the file

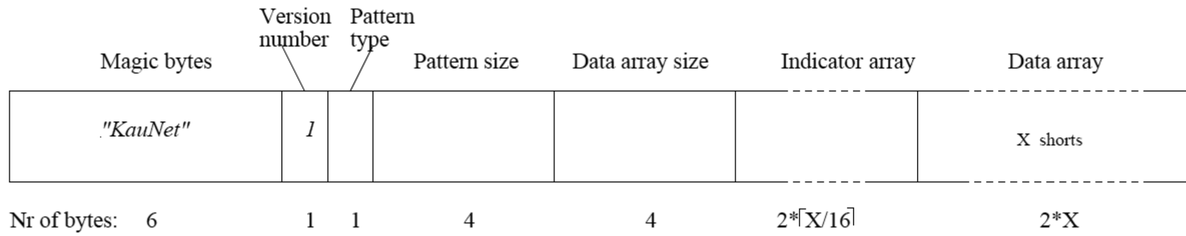

Figure 5.2: KauNet pattern structure (compressed format).

as a pattern file.

- Version number: The version number of the pattern. The only version currently defined is 1.
- Pattern type: The pattern type is composed by eight bits, where the two most significant bits indicate the mode of the pattern. There are three different modes available; data-driven (1), time-driven (2), and a combination (3) (possible for bit-error patterns). The lower six bits encode the pattern type: bit-errors (1), packet losses (2), delays (3), bandwidth (4), and triggers (5).
- Pattern size: The pattern size specifies the length of the pattern in kilobytes, packets, or milliseconds (see Table 5.1 for type of pattern and unit mapping). The size is an unsigned integer.
- Data array size: The data array size specifies the number of shorts in the data array (denoted by **X** in Figure 5.2).
- Indicator array: Each index in the array indicates if the short at the corresponding position in the data array is a run-length value or a data value. There are **X**/16 shorts in the indicator array.
- Data array: The values in the data array are either run-length values (for run-length encoding) or arbitrary data values (e.g. the bandwidth for bandwidth patterns or

trigger values for trigger patterns).

Pattern type	Size unit for data-driven mode	Size unit for time-driven mode
Bit-errors	Kilobyte	Kilobyte
Packet losses	Packets	Milliseconds
Delay changes	Packets	Milliseconds
Bandwidth changes	Packets	Milliseconds
Triggers	Packets	Milliseconds

Table 5.1: Type of pattern and unit mapping.

5.2.2 Pattern Generation

To support trigger patterns, minor modifications were implemented in *patt_gen* to allow the creation of a new value pattern. Trigger patterns can currently only be explicitly specified from the command line or in a file. In data-driven mode, the trigger patterns control after how many packets a trigger event is raised. For time-driven mode, the pattern specifies during which millisecond a trigger event is raised. The syntax for the creation of a trigger pattern is as follows:

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

`<filename>` The name of the output file.

`<mode>` Specifies the generation mode (data/time).

`<size>` The size of the generated pattern in the unit listed in Table 5.1, depending on the pattern type and mode.

`<position-value>` A comma separated list of tuples position-value separated by commas. The first position in the pattern is index 1.

`-f <infilename>` A filename of a text file containing numbers separated by either a comma or a row, describing positions and values. Two adjacent numbers form a position-value tuple.

5.3 Trigger Passing

The trigger passing mechanism is used to forward the events raised by the trigger patterns in KauNet and their associated information to both local and remote clients. To accomplish this, four primary components/modifications are implemented. The first is a modification to KauNet, allowing it to use functionality implemented in a kernel module loaded as a plugin to KauNet, as well as define the internal data that it wishes to pass with the event when it is raised. The second is the kernel module which KauNet is to use, called the KauNet Communication Module (KCM). It implements the functionality that allows clients to subscribe and unsubscribe to events, as well as receive events from KauNet and pass it on to the subscribers. The third is the Adaptation Layer Communication Module (ALCM) which is used by the client and provides an interface for communicating with the KCM. The module allows clients to connect to a KauNet machine to subscribe or unsubscribe to events, as well as receive event information. It also provides the functionality for parsing the received events to extract specific information. Last, a client application is necessary, called the Adaptation Layer. However, as this client is to provide the functionality desired by a performed experiment, no general adaptation layer can be implemented. A simple application was created to serve as an adaptation layer in the demonstration experiments included in Chapter 6, illustrating the use of the ALCM and interaction with the KauNet host.

5.3.1 KauNet

As mentioned in Section 4.2.1, the implementation of the trigger passing mechanism in the KauNet kernel was decided to be kept at a minimum to avoid too many modifications to the kernel and ease future modifications to both KauNet and the trigger passing. The trigger passing itself is externalized in a plugin module (the KCM which will be explained in greater detail in the next section). To interact with the module, KauNet defines a

function pointer type (*kcm_event_hook_t*) as a void function accepting a *kcm_event_data_t* struct as a parameter, and declares a static function pointer (*kcm_event_hook*) of the newly the defined type, initialized with a null value. Whenever an event is triggered in KauNet, KauNet will call the *pass_trigger()* function which checks to see if the pointer has been given a non-null value by an external KCM that has been loaded. If so, KauNet will store any internal data that it wishes to send in a *kcm_event_data_t* struct and pass this as an argument to the function referenced by the pointer. By default, the argument is limited to the trigger value (an integer), but can be modified fairly easily by modifying the contents of the struct. If the pointer has not been redefined or if it at any point has been reset to null, KauNet assumes that no KCM is currently loaded and the event is ignored as KauNet has nowhere to send it. Trigger patterns are both advanced and invoked at the same time. For time-driven trigger patterns, this takes place in the *dummynet_task()* function. For data-driven patterns, it takes place in the *dummynet_io()* function, prior to any packet loss taking place. See also the Dummynet implementation overview in Section 5.1.

5.3.2 KauNet Communication Module

The KauNet Communication Module (KCM) is a kernel module which provides the trigger passing mechanism, allowing events raised in KauNet to be forwarded to subscribing clients. The module consists of three parts; the load/unload handler (*kcm_handler*), the listener (*kcm_listener*), and the event broadcaster (*kcm_event_broadcaster*).

The load/unload handler has two operations to perform; create a separate thread for the listening and set the function pointer declared in KauNet (see previous section). When the module is loaded, the listener thread is created and the pointer is set so that it references the event broadcaster function. The module also has a flag which shows the current running state (set to *start* on load). When the module is unloaded the pointer is reset to null and the flag is set to *stop*. By changing the flag to *stop*, the thread can remove any allocated memory, setting the flag to *exit*, and terminate itself. The module waits for the flag to be

set to *exit* before it terminates, which enables a clean exit.

The purpose of the listener is to accept subscription and unsubscription requests from one or more adaptation layers. The listener creates a socket and binds it to a predefined port on all interfaces, on which it accepts requests from clients. For each request, the listener receives a packet containing the client IP address, the client port number and an additional flag, which specifies if it is a request for subscription (a non-zero value) or an unsubscription (a zero value). If the flag specifies a subscription, the client IP address and port number are added to a subscription list which stores all subscribing clients. If the flag specifies an unsubscription, the entry for the subscriber is removed from the list. During the unloading of KCM, all IP addresses and port numbers are removed from the subscriber list, and the socket is closed.

The purpose of the event broadcaster is to forward the events received from KauNet to all subscribing clients. For this, it uses the previously mentioned subscriber list to fetch the subscribers and forward the received trigger events. Each trigger event contains the trigger value and also a timestamp of the trigger occurrence, where the trigger value is received from the kernel and the timestamp is added in the module.

5.3.3 Adaptation Layer Communication Module

The Adaptation Layer Communication Module (ALCM) is a C-library providing an API for the adaptation layer. The ALCM interacts with KauNet via the KCM. The API includes the following functions for use by an arbitrary adaptation layer:

> *int subscribe(const char *ip, int port);*
>
> By calling this function, the adaptation layer will attempt to send a subscription request to the specified IP address (in the form x.x.x.x) and port number. If the receiver has a KCM loaded and listening on the specified port, the sender will be added as a subscriber and receive events. If 0 is provided as the port, the function will use the default port (1066) to connect. The function returns 0 if the request was

sent successfully, else -1 is returned.

int unsubscribe();

Assuming the client has previously subscribed, this function will send an unsubscription request to the same receiver. It returns 0 if the request was sent successfully, else -1.

void recv_event();*

This is a blocking function used to receive the next event from KauNet. It returns a void pointer to the received data if successful, else a null pointer.

int getDataSize();

This function returns the size of the event data struct in bytes.

In addition, the default ALCM provides two functions for parsing the received event data to retrieve specific information.

- *int getValue(void *data)*

  This function returns the trigger value (an integer) from the event data specified.

- *struct timeval getTimestamp(void *data)*

  This function returns the timestamp in the form of a timeval struct from the specified event data.

Depending on the needs of the experiment, KauNet or the KCM may need to be expanded to include additional information. By adding additional functions to the ALCM, it can be made aware of any other information included in the event data by KauNet or the KCM. If the event data is restructured, the *get* functions may require updating to extract the correct information. However, as long as they provide the same functionality, the ALCM can be updated while remaining backward compatible with older adaptation layers as the internal structure of the event data received is not relevant to the adaptation layer itself.

The provided ALCM is written as a userspace library for FreeBSD 7.2. Running it on other UNIX systems should require only minor changes to the code. More extensive modifications are necessary to adapt it for use in kernelspace.

5.3.4 Adaptation Layer

The adaptation layer is an application that utilizes the information sent by a KauNet host. The information sent is defined by KauNet (internal emulation information such as trigger value and pipe statistics) and the KCM (external information, such as timestamps). In the default implementation, this information includes the trigger value that is stored in the trigger pattern, and the timestamp of the event being raised. When creating an adaptation layer, the ALCM C-library needs to be included, which enables the adaptation layer to interact with a KauNet host via the functions described in Section 5.3.3. Depending on the needs of the experiment, a custom adaptation layer will need to be implemented to provide the desired functionality. Should any information in addition to the default (trigger value and timestamp) be required, this information should be included in KauNet and the KCM, and a parsing function should be implemented in the ALCM. A sample adaptation layer code is provided (see Appendix C.5) where the adaptation layer subscribes to a KauNet host, waits for events and prints out their content until manually terminated, at which points it sends an unsubscription request. This adaptation layer is used for the demonstration and verification in Chapter 6.

5.4 Summary

Described in this chapter are the implementation details for the trigger pattern added to KauNet, as well as the components implemented to allow triggers to be passed to local and remote subscribers. Figure 5.3 illustrates the complete system with its components and how they interact. Trigger patterns are a new type of value patterns, storing arbitrary

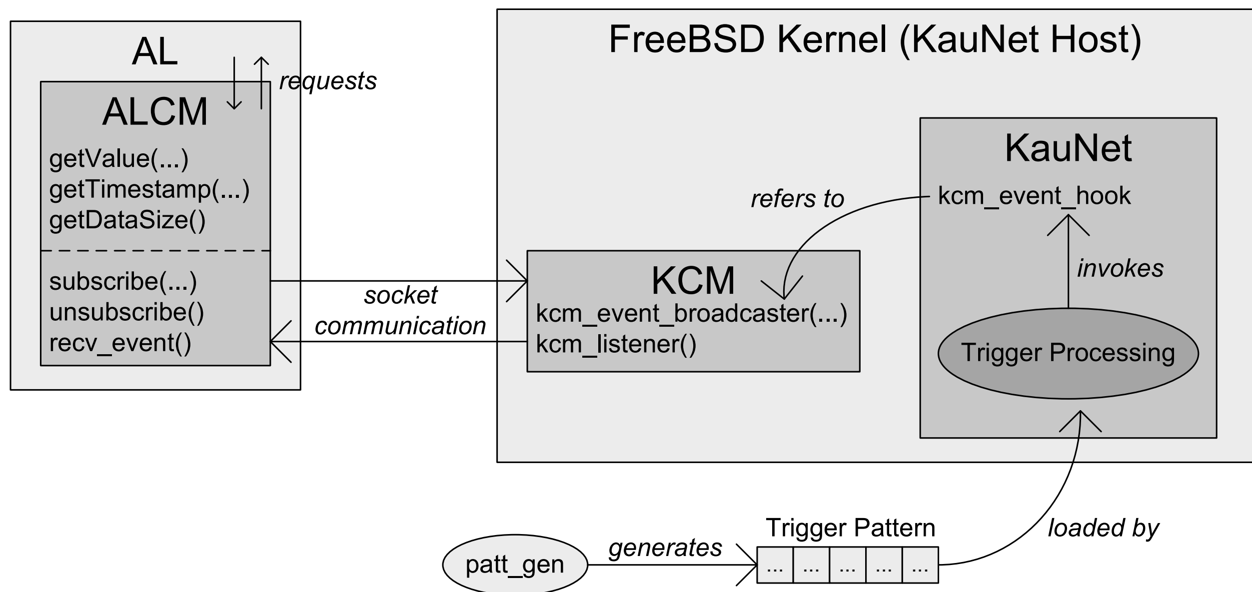

Figure 5.3: KauNet trigger passing implementation overview.

trigger values within them, that can be generated using a modified version of the pattern generation utility (*patt_gen*). Like other KauNet patterns, they can run in both time- and data-driven mode. When a trigger pattern is invoked in KauNet, an event is raised and KauNet calls a function via a function pointer (*kcm_event_hook*), passing the trigger value stored in the pattern as an argument (as well as any other internal information that may be necessary). The function pointer refers to a function (*kcm_event_broadcaster*) implemented in the KauNet Communication Module (KCM), a KauNet plugin kernel module loaded separately. When this function is called, the KCM will distribute the event via network sockets to any subscribing adaptation layers of which it is aware. The Adaptation Layer is an arbitrary application running either locally or on a remote host, acting as a receiver for the events. Using the interface functions provided by the Adaptation Layer Communication Module (ALCM), the adaptation layer can communicate with a KauNet host to send subscription/unsubscription requests and receive event information, as well as parse the received information.

Chapter 6

System Demonstration

In this chapter, a set of simple experiments are performed using trigger patterns in both time- and data-driven mode. The purposes of the experiments are to show that the implementation works as was intended, as well as illustrate how the trigger patterns can be used in KauNet.

6.1 Theory

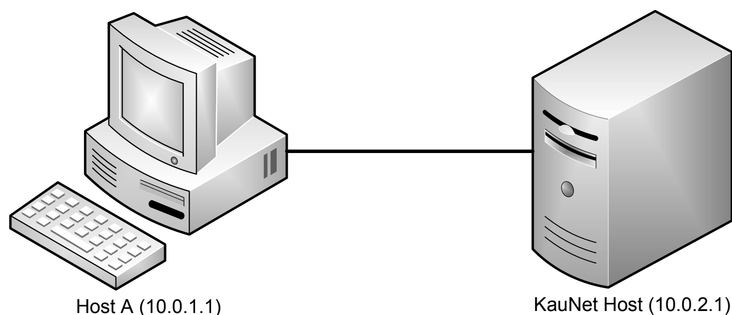

Figure 6.1: Test bed setup.

The experiments generate traffic from a FreeBSD machine to a KauNet emulator host, connected via a local network. Host A uses an adaptation layer to subscribe to events from the KauNet host, receiving a trigger value (an integer) and a timestamp of the event (local

time of the KauNet machine with microsecond precision). When an event is received, the adaptation layer simply prints the received value and timestamp. Traffic is generated using ping (ICMP) at a rate of two packets per second, and a total of 21 packets.

A total of two experiments will be performed, using two different sets of patterns:

1. A data-driven trigger pattern.
2. A time-driven bandwidth change pattern synchronized with a time-driven trigger pattern.

The data-driven trigger pattern is invoked at the arrival of the first packet, the ninth, the thirteenth and twenty-first (last) packet. For the time-driven patterns, both will be invoked at the fourth and sixth second of the experiment. In addition, the trigger pattern will also be invoked at the arrival of the first packet (zeroth second) and last packet (tenth second), signifying the start and end of the experiment. As the bandwidth change pattern is invoked for the first time, the available bandwidth of the pipe will be cut. The result is that all packets after the invocation are dropped until the second invocation, when the bandwidth is restored.

Packet (#)	1	2	3	4	5	6	7	8	9	10	11
Time (s)	0		1		2		3		4		5
Data trigger	1								2		
Time trigger	1								2		

Packet (#)	12	13	14	15	16	17	18	19	20	21
Time (s)		6		7		8		9		10
Data trigger		3								4
Time trigger		3								4

Table 6.1: Theoretical experiment results.

At the specified rate and number of packets, the results of the experiments can be predicted as illustrated in Table 6.1. Delays should be negligible in the experiment network, meaning the measured results should match the prediction fairly well. In addition, in the

second experiment, packet losses should occur from the fourth second to the sixth, during the period when there is no available bandwidth. At the specified rate, this results in a total of four lost packets

6.2 Method

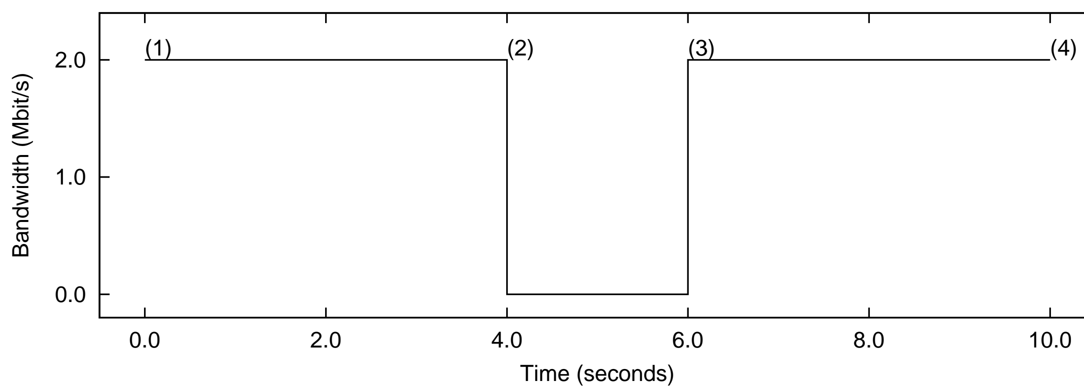

Figure 6.2: Experiment 1 setup.

For the purpose of these experiments, a sample adaptation layer has been implemented (see Appendix C.5) to receive and print the trigger values and timestamps, using the adaptation layer communication module (see Appendix C.4). For each experiment, traffic is generated using ping (ICMP) on a network connected machine (host A) to the KauNet emulator host (10.0.2.1). The patterns used are generated using *patt_gen*.

1. Data-driven trigger pattern.
   `./patt_gen -trig -pos trg_data.tp data 21 1,1,9,2,13,3,21,4`
   The trigger pattern is configured to be invoked at the arrival of the first, ninth, thirteenth and twenty-first packet. Each invocation is represented by a unique trigger value (1, 2, 3 and 4 respectively). These packets were selected to correspond to the invocations of the time-driven patterns (as illustrated in Table 6.1).

2. Time-driven trigger and bandwidth change pattern. The experiment is illustrated in Figure 6.2.

   ```
   ./patt_gen -bw -pos bw.bcp time 10000 4000,1,6000,2000
   ```

   The bandwidth change pattern will change the bandwidth from the initial bandwidth to no bandwidth (a bandwidth value of 1) at the fourth second, then restore the bandwidth to 2 Mbit/s at the sixth second.

   ```
   ./patt_gen -trig -pos trg_time.tp time 10000 1,1,4000,2,6000,3,10000,4
   ```

   The trigger pattern in synchronized with the bandwidth change pattern, being invoked at the fourth and sixth second. In addition, it is invoked as the experiment starts (first millisecond) and ends (tenth second). As in the data-driven trigger pattern, each invocation is represented by a unique trigger value.

The procedure for each experiment is almost identical, differing only in the patterns loaded into KauNet.

1. The experiments use a single pipe (ID 100) to filter outgoing ICMP traffic between the two machines. The pipe is added using IPFW.

   ```
   KauNet# ipfw add 1 pipe 100 icmp from 10.0.1.1 to 10.0.2.1 out
   ```

2. The pipe is configured to use an initial bandwidth of 2 Mbit/s and load the patterns used in the specific experiment using one of the following commands (depending on the experiment).

   1. `KauNet# ipfw pipe 100 config bw 2Mbit/s pattern trg_data.tp`
   2. `KauNet# ipfw pipe 100 config bw 2Mbit/s pattern bw.bcp pattern trg_time.tp`

3. The KauNet Communication Module (KCM) is loaded to enable trigger passing in KauNet to external adaptation layers.

   ```
   KauNet# kldload ./kcm.ko
   ```

4. The adaptation layer is loaded so that the remote machine can receive events from KauNet.

   `Host A# ./adaptationLayer`

   When loaded, the adaptation layer will automatically connect to the KauNet machine and start subscribing to events.

5. Host A uses ping to generate traffic at a rate of two packets per second and a total of 21 packets, using the KauNet machine as destination.

   `Host A# ping -c 21 -i 0.5  10.0.2.1`

6. (Optional) The KCM is unloaded and the adaptation layer is terminated using the SIGINT signal (Ctrl+C).

   `KauNet# kldunload kcm.ko`

6.3 Results

6.3.1 Experiment 1

Adaptation layer output (timestamps are written as seconds:microseconds):

```
Received value 1 at time 1260872513:989008 (12 bytes)
Received value 2 at time 1260872517:996164 (12 bytes)
Received value 3 at time 1260872520:000136 (12 bytes)
Received value 4 at time 1260872524:008111 (12 bytes)
```

The adaptation layer reports the arrival of the first packet when the first event (trigger value 1) arrives. Subsequent events arrive four, six and finally ten seconds later, which matches the expected arrival times of the selected packets.

6.3.2 Experiment 2

Adaptation layer output (timestamps are written as seconds:microseconds):

```
Received value 1 at time 1260871692:574073 (12 bytes)
Received value 2 at time 1260871696:581029 (12 bytes)
Received value 3 at time 1260871698:585016 (12 bytes)
Received value 4 at time 1260871702:592739 (12 bytes)
```

The start of the experiment (arrival of the first packet) is reported by the first event. Subsequent events arrive four, six and ten seconds later, as was expected from the created pattern.

Ping results:

```
PING 10.0.2.1 (10.0.2.1): 56 data bytes
64 bytes from 10.0.2.1: icmp_seq=1 ttl=64 time=1.261 ms
64 bytes from 10.0.2.1: icmp_seq=2 ttl=64 time=0.317 ms
64 bytes from 10.0.2.1: icmp_seq=3 ttl=64 time=0.310 ms
64 bytes from 10.0.2.1: icmp_seq=4 ttl=64 time=0.323 ms
64 bytes from 10.0.2.1: icmp_seq=5 ttl=64 time=0.308 ms
64 bytes from 10.0.2.1: icmp_seq=6 ttl=64 time=0.319 ms
64 bytes from 10.0.2.1: icmp_seq=7 ttl=64 time=0.322 ms
64 bytes from 10.0.2.1: icmp_seq=8 ttl=64 time=0.325 ms
64 bytes from 10.0.2.1: icmp_seq=13 ttl=64 time=0.257 ms
64 bytes from 10.0.2.1: icmp_seq=14 ttl=64 time=0.333 ms
64 bytes from 10.0.2.1: icmp_seq=15 ttl=64 time=0.334 ms
64 bytes from 10.0.2.1: icmp_seq=16 ttl=64 time=0.334 ms
64 bytes from 10.0.2.1: icmp_seq=17 ttl=64 time=0.337 ms
64 bytes from 10.0.2.1: icmp_seq=18 ttl=64 time=0.334 ms
64 bytes from 10.0.2.1: icmp_seq=19 ttl=64 time=0.339 ms
```

```
64 bytes from 10.0.2.1: icmp_seq=20 ttl=64 time=0.330 ms
64 bytes from 10.0.2.1: icmp_seq=21 ttl=64 time=0.332 ms

--- 10.0.2.1 ping statistics ---
21 packets transmitted, 17 packets received, 19.0% packet loss
```

The ping results show loss of the ninth through twelfth packet for a total of four lost packets, as was expected given the loaded bandwidth pattern.

6.4 Conclusions

The achieved results of both experiments are consistent with the expected results. In the second experiment, the bandwidth change pattern causes loss of packets nine through twelve (a total of four packets), corresponding to the two second window with no available bandwidth. The trigger patterns also behave as expected, raising an initial event (1) as the first packet arrives and provides a baseline timestamp for the remaining events. The remaining events are received at the expected points; four, six and ten seconds later.

According to these results, trigger patterns in both time- and data-driven mode work as intended. Perhaps of note is the slight drift in the timestamps, slightly under two milliseconds per second on average, although there is insufficient data to draw any definite conclusion. Even if it is not a coincidence in these specific tests, it may be a problem with KauNet or Dummynet in general, as opposed to the implementation of the trigger passing mechanism.

Chapter 7

Conclusions & Future Work

In this thesis a new KauNet pattern has been introduced. The new pattern, called trigger pattern, enables information to be passed out from the kernel to an application. This new functionality enables the previous on-demand access of information (i.e. get emulation statistics via IPFW) to be provided directly when occurring. The trigger passing mechanism can also provide cross-layer information to network connected computers or provide real-time logging of the emulation.

The design for the trigger pattern and its implementation has been described in detail, as well as the two communication modules needed for the trigger passing mechanism; the KauNet Communication Module (KCM) and the Adaptation Layer Communication Module (ALCM). The KCM handles the trigger passing from the kernel to the ALCM, and the ALCM forwards the trigger information to an arbitrary adaptation layer. In addition to the trigger pattern and the modules, a sample adaptation layer has been supplied to show how to use the ALCM to interact with the KCM and KauNet. The sample adaptation layer was tested and performed as expected. The setup and output are provided in Chapter 6.

The formal requirements for the trigger patterns were, primarily, to be able to send event information from KauNet to an adaptation layer at least once every millisecond. Two

proposed communication mechanisms, signals and sockets, have been evaluated.While all the tested mechanisms performed sufficiently, the method chosen was sockets. The reason for this was implementation reasons; network sockets allow events to be sent directly instead of relying on an intermediate layer, reducing the complexity of the mechanism.

Another requirement was to send information (the trigger value) with every event from the kernel. Sockets were able to send (reliably without losses) at the rate of at least 10 events per millisecond, using an event payload size of up to 1000 bytes.

The results from the evaluation shows that the socket communication is able to send at a rate of at least 10 events/millisecond to a remote receiver. By using a Gigabit Ethernet card instead of a 100 Mbit/s card, the rate may be increased further. To enable reliable use of trigger patterns in data-driven mode (one event per packet) in a network with a high packet rate, aggregation of the events may be needed. By design, aggregation is possible to implement in the KCM without modifying the kernel source code, as all changes are made in the modules.

7.1 Future Work

In the current implementation of the pattern generation utility triggers can only be created by specifying the positions and values by hand (or from a file). A new feature would be to create a trigger pattern from an existing pattern (bandwidth, delay, packet loss, bit-error). The difficulty of creating a trigger pattern from another pattern, is to specify when or where the trigger should occur. For example, a simple trigger pattern can be created from a bandwidth pattern, by generating a trigger when the bandwidth is changing. A more difficult trigger pattern would be to create a trigger when the bandwidth exceeds a specified threshold. For the second example, a secondary script language might be needed, to simplify the trigger pattern creation.

References

[1] J. Postel, “Transmission Control Protocol,” RFC 793, Internet Engineering Task Force, September 1981.

[2] J. Padhye, V. Firoiu, D. F. Towsley, and J. F. Kurose, “Modeling TCP reno performance: a simple model and its empirical validation,” *IEEE/ACM Trans. Netw.*, vol. 8, no. 2, pp. 133–145, 2000.

[3] M. Allman and A. Falk, “On the effective evaluation of TCP,” *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 5, pp. 59–70, 1999.

[4] V. Paxson and S. Floyd, “Why we don't know how to simulate the Internet,” in *WSC '97: Proceedings of the 29th conference on Winter simulation*, (Washington, DC, USA), pp. 1037–1044, IEEE Computer Society, 1997.

[5] L. R. Marta Carbone, “Dummynet Revisited,” tech. rep., Dipartimento di Ingegneria dell'Informazione Universita di Pisa, May 2009. `http://info.iet.unipi.it/ luigi/papers/20090531-ccr-dummynet.pdf`.

[6] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu, and D. Zappala, “Improving Simulation for Network Research,” Tech. Rep. 99-702b, University of Southern California, March 1999. revised September 1999, to appear in IEEE Computer.

[7] “The Network Simulator - ns-2,” October 2009. `http://www.isi.edu/nsnam/ns/`.

[8] “The ns-3 network simulator,” October 2009. `http://www.nsnam.org/`.

[9] “Iperf,” October 2009. `http://sourceforge.net/projects/iperf/`.

[10] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker, “Scalability and accuracy in a large-scale network emulator,” *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 271–284, 2002.

[11] “Dummynet,” October 2009. `http://info.iet.unipi.it/~luigi/dummynet/`.

[12] J. Garcia, P. Hurtig, and A. Brunstrom, "Kaunet," October 2009. `http://kaunet.sourceforge.net/`.

[13] L. Rizzo, "Dummynet and forward error correction," in *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 31–31, USENIX Association, 1998.

[14] B. Brade, D. Clark, J. Crowcroft, b. Davie, S.Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramkrishnan, S. Shenker, J. Wroclawski, and L. Zhang, "Recommendations on Queue Management and Congestion Avoidance in the Internet," RFC 2309, April 1998.

[15] J. C. R. Bennett and H. Zhang, "Hierarchical packet fair queueing algorithms," in *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, (New York, NY, USA), pp. 143–156, ACM, 1996.

[16] J. C. R. Bennett and H. Zhang, "WF2Q: Worst-case Fair Weighted Fair Queueing," in *IEEE IN-FOCOM '96*, (San Francisco, CA), March 1996.

[17] The FreeBSD Foundation, "The FreeBSD Handbook," September 2009. `http://www.freebsd.org/doc/en/books/handbook`.

[18] J. Postel, "Internet Control Message Protocol," RFC 792, Internet Engineering Task Force, September 1981.

[19] J. Garcia, P. Hurtig, and A. Brunstrom, "KauNet: Design and Usage," Technical Report 2008:59, Karlstad University, December 2008.

[20] E. N. Gilbert, "Capacity of a Burst-Noise Channel," tech. rep., Bell System Technical Journal, September 1960.

[21] E. O. Elliot, "Estimates of error rates for codes on burst-noise channels," tech. rep., Bell System Technical Journal, September 1963.

[22] X. Yu, "Improving TCP performance over mobile ad hoc networks by exploiting cross-layer information awareness," in *MobiCom '04: Proceedings of the 10th annual international conference on Mobile computing and networking*, (New York, NY, USA), pp. 231–244, ACM, 2004.

[23] C. Mbarushimana, A. Shahrabi, and T. Buggy, "A cross-layer support for TCP enhancement in qos-aware mobile ad hoc networks," in *MSWiM '08: Proceedings of the 11th international symposium on Modeling, analysis and simulation of wireless and mobile systems*, (New York, NY, USA), pp. 10–17, ACM, 2008.

[24] J. Papandriopoulos, S. Dey, and J. Evans, "Optimal and distributed protocols for cross-layer design of physical and transport layers in MANETs," *IEEE/ACM Trans. Netw.*, vol. 16, no. 6, pp. 1392–1405, 2008.

[25] M. Allman, V. Paxson, and E. Blanton, "TCP congestion control," RFC 5681, Internet Engineering Task Force, September 2009.

[26] S. Floyd, T. Henderson, and A. Gurtov, "The NewReno Modification to TCP's Fast Recovery Algorithm," RFC 3782, Internet Engineering Task Force, April 2004.

[27] A. Karnik and A. Kumar, "Performance of TCP congestion control with explicit rate feedback," *IEEE/ACM Trans. Netw.*, vol. 13, no. 1, pp. 108–120, 2005.

[28] S. Floyd, M. Allman, A. Jain, and P. Sarolahti, "Quick-start for TCP and IP," RFC 4782, Internet Engineering Task Force, January 2007.

[29] S. Floyd, M. Handley, J. Padhye, and J. Widmer, "Equation-based congestion control for unicast applications," in *SIGCOMM '00: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, (New York, NY, USA), pp. 43–56, ACM, 2000.

[30] "TCP westwood home page," October 2003. `http://www.cs.ucla.edu/NRL/hpi/tcpw/`.

[31] C. Casetti, M. Gerla, S. Mascolo, M. Y. Sanadidi, and R. Wang, "TCP westwood: end-to-end congestion control for wired/wireless networks," *Wirel. Netw.*, vol. 8, no. 5, pp. 467–479, 2002.

[32] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, "TCP westwood: Bandwidth estimation for enhanced transport over wireless links," in *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, (New York, NY, USA), pp. 287–297, ACM, 2001.

[33] S. R. Walli, "The POSIX family of standards," *StandardView*, vol. 3, no. 1, pp. 11–17, 1995.

[34] J. Postel, "User Datagram Protocol," RFC 768, Internet Engineering Task Force, August 1980.

[35] The FreeBSD Foundation, "The FreeBSD Project," September 2009. `http://www.freebsd.org`.

[36] F. Pohlmann, "Why FreeBSD," September 2009. `http://www.ibm.com/developerworks/opensource/library/os-freebsd/`.

[37] SpreadBSD, "FreeBSD," September 2009. `http://www.spreadbsd.org/?q=freebsd`.

[38] N. Saers, "A project model for the FreeBSD Project," September 2009. http://www.freebsd.org/doc/en_US.ISO8859-1/books/dev-model/.

Appendix A

IPC Evaluation

During the trigger passing mechanism evaluation, two computers were used. The specifications of these computers can be read below, where the first computer is the main computer with the KauNet kernel installed. The second computer is only used during the network socket tests. See Section 4.2.2 for details about the evaluation.

A.1 Test Machine Specifications

Processor:	Intel® Pentium® 4 CPU 2.80GHz
Memory:	512MB DDR SDRAM System Memory
Operating System:	FreeBSD 7.2-RELEASE (KauNet kernel)
Hard drive:	40GB (WD400BB)
Network controller:	3Com® 3c905C-TX Fast Etherlink XL
	Intel® 82559 Pro/100 Ethernet
	Intel® PRO/1000 Network Connection 6.9.6

Table A.1: IPC test bed specification, KauNet host.

Processor:	Intel® Pentium® 4 CPU 2.80GHz
Memory:	1024MB DDR SDRAM System Memory
Operating System:	FreeBSD 7.2-RELEASE (GENERIC kernel)
Hard drive:	80GB (WD800BB)
Network controller:	3Com® 3c905C-TX Fast Etherlink XL
	Intel® 82559 Pro/100 Ethernet
	Intel® PRO/1000 Network Connection 6.9.6

Table A.2: IPC test bed specification, receiving host.

Appendix B

FreeBSD

FreeBSD [35] is an operating system and the platform on which Dummynet and KauNet are implemented. FreeBSD was first developed in 1993, based on the BSD branch of the UNIX kernel [17]. While it is not a UNIX operating system, it is considered "UNIX-like" and work in a similar manner and uses UNIX-compliant system APIs [36]. It also provides optional binary compatibility with other UNIX and UNIX-like systems, such as Linux, enabling most applications developed for another system to run on FreeBSD as well [17], although some applications may only be partially or not at all supported. Unlike operating systems such as Linux where distributions are packaged by selecting the kernel and applications separately, FreeBSD is a complete OS, where the kernel, drivers and user applications are all developed and packaged together for each version [37].

Development of FreeBSD is done using several versions in parallel [38]. Each major version receives a *-STABLE* branch, representing the primary distribution. From this branch, new releases are created, supporting additional or modified features. If the new features are deemed stable and developed enough, they may then be backported into *-STABLE*. The *-CURRENT* branch represents the latest version in development not yet completed, similar to a beta. At the time of writing, the most recent release is 7.2, released in May 2009, which is also the version for which the thesis project is implemented.

Appendix C

Source code

This appendix contains the source code, or the difference in the same, for the FreeBSD kernel (ip_dummynet), the communication modules, the sample adaptation layer, the pattern generation tool, and IPFW.

C.1 File Difference Output

Below is a brief description of the output seen in Section C.2, Section C.6 and Section C.7. For a full documentation, run *man diff* or *info diff* from a command line.

A minus sign in front of a row indicates a removed or replaced line in the source code. A plus sign in front of a row indicates a replacement (of a previously removed line) or an inserted line. For each deletion/change, a 20 line context (ten lines before and after) are showed to aid in the understanding of the modification.

The numbers surrounded by two at-signs (on each side) are the starting line number and the number of lines the change applies to for each file; the minus sign indicates the original file and the plus sign indicates the modified file (with trigger patterns). For example, if a line has been inserted on line 15, the line should state `@@ -5,20 +5,21 @@`. For the original file, the starting line is 5 (line 15 - 10 context lines) and the number of lines is 20

(2 * 10 context lines). For the modified file, the starting line is still 5 as in the original file, but the number of lines is 21 (2 * 10 context lines + 1 inserted line).

C.2 FreeBSD Kernel (with KauNet)

The following output shows the difference between the original ip_dummynet source code (FreeBSD 7.2 with KauNet) and the same source code with trigger pattern implemented. See Section C.1 for a description of the output. In addition to the modifications in the ip_dummynet files, the defined constant *MAX_PATTERNS* is incremented to five, to make room for the trigger pattern.

C.2.1 ip_dummynet.h

```
@@ -373,11 +373,23 @@
 ip_dn_claim_rule(struct mbuf *m)
 {
  struct m_tag *mtag = m_tag_find(m, PACKET_TAG_DUMMYNET, NULL);
  if (mtag != NULL) {
   mtag->m_tag_id = PACKET_TAG_NONE;
   return (((struct dn_pkt_tag *)(mtag+1))->rule);
  } else
   return (NULL);
 }
 #endif
+
+#ifdef KAUNET /* KAUNET++ */
+/* Struct used for sending data to the KCM via the hook */
+struct kcm_event_data_t {
+ int value;
```

```
+};
+
+/*  A typedef for the kernel module function pointer used to inform
+ the kernel communication module that an event has taken place */
+typedef void (*kcm_event_hook_t)(struct kcm_event_data_t data);
+#endif /* KAUNET++ */
+
 #endif /* _IP_DUMMYNET_H */
```

C.2.2 ip_dummynet.c

```
@@ -245,20 +245,26 @@

 static void dummynet(void *);
 static void dummynet_flush(void);
 static void dummynet_send(struct mbuf *);
 void  dummynet_drain(void);
 static ip_dn_io_t dummynet_io;
 static void dn_rule_delete(void *);

 #ifdef KAUNET
 static int delete_pipe(struct dn_pipe *p);
+
+/* Trigger passing function */
+void pass_trigger(struct dn_pipe* pipe);
+
+/* KCM module hook */
+static kcm_event_hook_t kcm_event_hook = NULL;
 #endif /* KAUNET */
```

```
 /*
  * Heap management functions.
  *
  * In the heap, first node is element 0. Children of i are 2i+1 and 2i+2.
  * Some macros help finding parent/children so we can optimize them.
  *
  * heap_init() is called to expand the heap when needed.
@@ -947,27 +953,39 @@
       *
       * 1) any_active() checks if the pipe contains any patterns and if they
       * are allowed to be traversed yet. If so, increase the number of "calls" to
       * the pipe
       *
       * 2) should_advance() checks if the pattern is time-driven and appropriate
       * to use in a time-driven mode. The second condition checks if the number
       * of pipe calls equals 1 ms
       *
       * 3) If a certain pattern is active, it will be advanced
+   *
+   * 4) If a trigger pattern is active and has a trigger value set, transmit the
+   * event if the communication module is loaded. Increment invocations either way
+   *
       */
    if (any_active(pipe))
     pipe->calls++;
    int bytes_to_consume;
    for (j = 0; j < MAX_PATTERNS; j++) {
```

```
     if (should_advance(pipe, j) && (pipe->calls * 1000 == hz)) {
      switch (j) {
+      case TRIGGER:
+       advance_valuepattern(pipe->patterns[TRIGGER]);
+       pipe->pattern_pos[TRIGGER]++;
+       if (pipe->patterns[TRIGGER]->trigger_value) {
+        pass_trigger(pipe);
+        pipe->invocations[TRIGGER]++;
+       }
+       break;
       case BITERROR:
        bytes_to_consume = pipe->bandwidth / (8*hz);
        pipe->patterns[BITERROR]->bit_overflow += pipe->patterns[BITERROR]->\
bandwidth % (8*hz);
        if (pipe->patterns[BITERROR]->bit_overflow > (8*hz)) {
         bytes_to_consume++;
         pipe->patterns[BITERROR]->bit_overflow -= (8*hz);
        }
        advance_biterror(pipe->patterns[BITERROR], bytes_to_consume);
        pipe->pattern_pos[BITERROR] += (bytes_to_consume * 8);
        break;
@@ -1459,20 +1477,31 @@
  if (fs->flags_fs & DN_IS_RED && red_drops(fs, q, len))
   goto dropit;

 #ifdef KAUNET
  /*
   * As a packet is entering the emulator, activate all patterns
   * present. Also, if a packet loss pattern is active, advance
```

```
   * it and drop the packet if appropriate
   */
     if (is_pipe) {
+  if (pipe->patterns[TRIGGER]) {
+   pipe->patterns[TRIGGER]->defer_start = 0;
+   if (!pipe->patterns[TRIGGER]->timedriven) {
+    advance_valuepattern(pipe->patterns[TRIGGER]);
+    pipe->pattern_pos[TRIGGER]++;
+    if (pipe->patterns[TRIGGER]->trigger_value) {
+     pass_trigger(pipe);
+     pipe->invocations[TRIGGER]++;
+    }
+   }
+  }
  if (pipe->patterns[BITERROR]) {
      pipe->patterns[BITERROR]->defer_start = 0;
  }
  if (pipe->patterns[PACKET_LOSS]) {
      pipe->patterns[PACKET_LOSS]->defer_start = 0;
      if (!pipe->patterns[PACKET_LOSS]->timedriven) {
   advance_packetloss(pipe->patterns[PACKET_LOSS]);
          pipe->pattern_pos[PACKET_LOSS]++;
      }
      if (pipe->patterns[PACKET_LOSS]->lose_packet == 1) {
@@ -2543,13 +2572,25 @@
   break ;
  }
  return 0 ;
 }
```

```
 static moduledata_t dummynet_mod = {
  "dummynet",
  dummynet_modevent,
  NULL
 };
+
+#ifdef KAUNET
+void pass_trigger(struct dn_pipe *pipe)
+{
+ if (kcm_event_hook != NULL) {
+  struct kcm_event_data_t data;
+  data.value = pipe->patterns[TRIGGER]->trigger_value;
+  kcm_event_hook(data);
+ }
+}
+#endif /* KAUNET */
+
 DECLARE_MODULE(dummynet, dummynet_mod, SI_SUB_PROTO_IFATTACHDOMAIN, SI_ORDER_ANY);
 MODULE_DEPEND(dummynet, ipfw, 2, 2, 2);
 MODULE_VERSION(dummynet, 1);
```

C.3 KauNet Communication Module

The KauNet Communication Module (described in Section 4.2.3 and Section 5.3.2) is composed by two parts; the header file and the source code file. The header file contains definitions used by the module, while the source code file contains the implementation for the trigger passing functionality.

C.3.1 kcm.h

```
1  #ifndef __KCM_H__
2  #define __KCM_H__
3
4  #include <sys/param.h>   /* kernel.h defines */
5  #include <sys/module.h>
6  #include <sys/kernel.h>  /* types in module init */
7  #include <sys/systm.h>   /* uprintf, optional */
8  #include <sys/malloc.h>  /* malloc macros */
9  #include <sys/types.h>
10 #include <sys/errno.h>
11 #include <sys/kthread.h> /* kthread_create */
12 #include <sys/unistd.h>  /* RFNOWAIT */
13 #include <machine/stdarg.h> /* valist */
14 #include <sys/proc.h>    /* curthread, proc0 */
15 #include <sys/lock.h>  /* LOCK_FILE, LOCK_LINE */
16 #include <sys/mutex.h>   /* mtx_xxx */
17 #include <sys/sched.h>   /* sched_add */
18 #include <sys/queue.h>  /* slist */
19
20 //socket
21 #include <sys/socket.h>
22 #include <sys/libkern.h>
23 #include <netinet/in.h>  /* sockaddr_in */
24 #include <sys/socketvar.h> /* so-functions */
25 #include <sys/uio.h>  /* struct uio */
26
27
28 static int tstate;
```

```
29 #define T_START 0
30 #define T_STOP 1
31 #define T_EXIT 2
32
33 #define PORT 1066
34 #define UNSUBSCRIBE 0
35 #define DEBUG
36
37 struct kcm_event_data_t {
38  int value;
39 };
40
41 struct kcm_send_event{
42  int value;
43  struct timeval timestamp;
44 } __attribute__((packed));
45
46 struct subscription_data_t{
47  int subscribe; /* 0: unsubscribe, 1: subscribe */
48 };
49
50 struct socket *so;
51
52 /* SLIST stuff */
53 struct slist_entry {
54  struct sockaddr_in subscriber;
55  SLIST_ENTRY(slist_entry) slist_entries;
56 };
57 SLIST_HEAD(, slist_entry) head;
```

```
58 MALLOC_DEFINE(M_SLIST, "slist", "slist_entry");
59
60 /* Stuff for kernel hook */
61 typedef void (*kcm_event_hook_t)(struct kcm_event_data_t data);
62 extern kcm_event_hook_t kcm_event_hook;
63
64 #endif //__KCM_H_
```

C.3.2 kcm.c

```
1  #include "kcm.h"
2
3  /* Prototypes */
4  void kcm_event_broadcaster(struct kcm_event_data_t data);
5  void kcm_listener(void);
6  void kcmExit(void);
7
8  void kcmExit(void) {
9   tstate = T_EXIT;
10  kthread_exit(0);
11 }
12
13 /* Need socreate and sobind from kcm_listener to be run before sending
14  any packets, but no packets are sent before any subscribers
15  received (i.e. socreate and sobind have been run)
16
17  1. socreate and sobind
18  2. soreceive => subscribers added to list
19  3. kaunet_event send packets to subscribers
```

```
20 */
21 void kcm_event_broadcaster(struct kcm_event_data_t data)
22 {
23  int ret;
24  struct iovec aiov;
25  struct uio auio;
26  struct slist_entry *item;
27  struct kcm_send_event message;
28  struct timeval timestamp;
29 
30  microtime(&timestamp);
31 
32  /* auio */
33  auio.uio_iov = &aiov;
34  auio.uio_iovcnt = 1;
35  auio.uio_offset = 0;
36  auio.uio_rw = UIO_WRITE; /* Read from socket */
37  auio.uio_segflg = UIO_SYSSPACE;
38  auio.uio_td = curthread;
39 
40  SLIST_FOREACH(item, &head, slist_entries) {
41   //Add the data to send
42   message.value = htonl(data.value);
43   message.timestamp.tv_sec = htonl(timestamp.tv_sec);
44   message.timestamp.tv_usec = htonl(timestamp.tv_usec);
45   #ifdef DEBUG
46    printf("Sending value: %d %ld:%ld\n",
47     data.value,
48     (long int)timestamp.tv_sec,
```

```
49      (long int)timestamp.tv_usec);
50    #endif
51
52    /* Data to send */
53    aiov.iov_base = &message;
54    aiov.iov_len = sizeof(struct kcm_send_event);
55    auio.uio_resid = sizeof(struct kcm_send_event);
56
57    ret = sosend(so, (struct sockaddr*)&item->subscriber, &auio,
58     0, NULL, 0, curthread);
59    if (ret != 0) {
60     printf("sosend, ret: %d\n", ret);
61    }
62   }
63  }
64
65  void kcm_listener(void) {
66   int ret, flag = MSG_DONTWAIT;
67   struct iovec aiov;
68   struct uio auio;
69   struct sockaddr_in sain;
70   struct sockaddr_in *fsain;
71   struct subscription_data_t controlData;
72   /* tailq list */
73   int slist_size = sizeof(struct slist_entry);
74   struct slist_entry *item;
75   SLIST_INIT(&head);
76
77   ret = socreate(AF_INET, &so, SOCK_DGRAM, 0, curthread->td_ucred, curthread);
```

```
78  if (ret != 0) {
79   printf("socreate, ret: %d\n", ret);
80   kcmExit();
81  }
82
83  sain.sin_family = AF_INET;
84  sain.sin_port = htons(PORT);
85  sain.sin_addr.s_addr = INADDR_ANY;
86  sain.sin_len = sizeof(sain);
87
88  ret = sobind(so, (struct sockaddr*)&sain, curthread);
89  if (ret != 0){
90   printf("sobind, ret: %d\n", ret);
91   kcmExit();
92  }
93  auio.uio_iov = &aiov;
94  auio.uio_iovcnt = 1;
95  auio.uio_offset = 0;
96  auio.uio_rw = UIO_READ; /* Read from socket */
97  auio.uio_segflg = UIO_SYSSPACE;
98  auio.uio_td = curthread;
99
100 while (tstate == T_START) {
101  /* Data to receive */
102  aiov.iov_base = &controlData;
103  aiov.iov_len = sizeof(struct subscription_data_t);
104  auio.uio_resid = sizeof(struct subscription_data_t);
105
106  /* wait for packet to arrive with ip address and port */
```

```
107  ret = soreceive(so, (struct sockaddr**)&fsain, &auio, 0, NULL, &flag);
108  switch(ret){
109   case EAGAIN:
110   case 0:
111    /* If packet received */
112    if (auio.uio_resid != sizeof(controlData)) {
113     if (controlData.subscribe != UNSUBSCRIBE) {
114      MALLOC(item, struct slist_entry*, slist_size, M_SLIST, M_NOWAIT);
115      item->subscriber = *fsain;
116      SLIST_INSERT_HEAD(&head, item, slist_entries);
117     } else if (controlData.subscribe == UNSUBSCRIBE) {
118      SLIST_FOREACH(item, &head, slist_entries) {
119       if (item->subscriber.sin_port == fsain->sin_port &&
120        item->subscriber.sin_addr.s_addr == fsain->sin_addr.s_addr) {
121        SLIST_REMOVE(&head, item, slist_entry, slist_entries);
122        FREE(item, M_SLIST);
123       }
124      }
125     }
126     #ifdef DEBUG
127      printf("%subscription from %s:%d\n",
128       (controlData.subscribe != UNSUBSCRIBE?"S":"Uns"),
129       inet_ntoa(fsain->sin_addr),
130       ntohs(fsain->sin_port));
131     #endif
132    }
133    break;
134   default:
135    printf("soreceive, ret: %d\n", ret);
```

```
136     break;
137    }
138    pause(NULL, 100);
139   }
140
141   ret = soclose(so);
142   if (ret != 0) {
143    uprintf("soclose, ret: %d\n", ret);
144   }
145
146   /* Remove all subscribers */
147   while ((item = SLIST_FIRST(&head))) {
148    #ifdef DEBUG
149     printf("Removing client %s:%d\n",
150      inet_ntoa(item->subscriber.sin_addr),
151      item->subscriber.sin_port);
152    #endif
153    SLIST_REMOVE(&head, item, slist_entry, slist_entries);
154    FREE(item, M_SLIST);
155   }
156   kcmExit();
157  }
158
159  static int kcm_handler(module_t mod, int op, void *arg){
160   int err = 0;
161   switch (op){
162    case MOD_LOAD:
163     tstate = T_START;
164     kcm_event_hook = &kcm_event_broadcaster;
```

```
165     kthread_create((void *)kcm_listener, NULL, NULL, RFNOWAIT, 0, "KauNetCM");
166     printf("KauNet Communication Module Loaded.\n");
167     break;
168    case MOD_UNLOAD:
169     kcm_event_hook = NULL;
170     if (tstate == T_START){
171      tstate = T_STOP;
172     }
173     while (tstate != T_EXIT){
174      tsleep(curthread, PDROP, "KCM tsleep", 2*hz);
175     }
176     printf("KauNet Communication Module Unloaded.\n");
177     break;
178    default:
179     err = EOPNOTSUPP;
180     break;
181   }
182   return err;
183  }
184
185  static moduledata_t kcm_data= {
186   "kcm",
187   kcm_handler,
188   0
189  };
190
191  MODULE_VERSION(kcm, 1);
192  DECLARE_MODULE(kcm, kcm_data, SI_SUB_KLD, SI_ORDER_ANY);
193
```

```
194
```

C.4 Adaptation Layer Communication Module

The Adaptation Layer Communication Module (described in Section 4.2.5 and Section 5.3.3) is composed by two parts; the API and the implementation of the API functions. The implementation of the API depends on how the KauNet Communication Module is implemented, e.g. how a subscription is made or how the received message is constructed (value and timestamp).

C.4.1 alcm.h

```
1  #ifndef __ALCM_H__
2  #define __ALCM_H__
3
4  #include <stdlib.h>
5  #include <sys/types.h>
6  #include <sys/socket.h>
7  #include <stdio.h>
8  #include <netinet/in.h>
9  #include <arpa/inet.h>
10
11 #define PORT 1066
12 #define SUBSCRIBE 1
13 #define UNSUBSCRIBE 0
14
15 /*
16  * Start subscribing to KCM at address ip, listening on port
17  * Returns 0 if successful, else -1
```

```
18  */
19 int subscribe(const char* ip, int port);
20
21 /*
22  * Unsubscribe from KCM currently connected to
23  * Returns 0 if successful or not currently connected, else -1
24  */
25 int unsubscribe();
26
27 /*
28  * Blocking receive function for receiving event data
29  * Returns void ptr to received data, or NULL on failure
30  */
31 void* recv_event();
32
33 /*
34  * Parse event data and retrieve trigger value
35  * Returns trigger value as int
36  */
37 int getValue(const void* data);
38
39 /*
40  * Parse event data and retrieve timestamp
41  * Returns timestamp as struct timeval
42  */
43 struct timeval getTimestamp(const void* data);
44
45 /*
46  * Get the size of the event data struct
```

```
47  * Returns size in bytes
48  */
49 int getDataSize();
50
51
52 struct subscription_data_t{
53  int subscribe;
54 };
55
56 /* The structure of the struct KCM sends */
57 struct kcm_receive_event {
58  int value;
59  struct timeval timestamp;
60 };
61
62 #endif /* __ALCM_H__ */
```

C.4.2 alcm.c

```
1 #include "alcm.h"
2
3 int sockfd = 0;
4 struct sockaddr_in server;
5
6 int subscribe(const char* ip, int port)
7 {
8  int ret;
9  struct subscription_data_t connectData;
10
```

```
11  if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0){
12   perror("socket");
13   return -1;
14  }
15
16  server.sin_family = AF_INET;
17  server.sin_port = (port==0?htons(PORT):htons(port));
18  if (inet_aton(ip, &server.sin_addr) == 0){
19   perror("inet_aton");
20   return -1;
21  }
22
23  connectData.subscribe = htonl(SUBSCRIBE);
24  ret = sendto(sockfd, &connectData, sizeof(struct subscription_data_t),
25   0, (struct sockaddr*)&server, sizeof(struct sockaddr_in));
26  if (ret == -1) {
27   perror("sendto - subscribe");
28   return -1;
29  }
30
31  return 0;
32 }
33
34 int unsubscribe()
35 {
36  if (sockfd != 0) {
37   int ret;
38   struct subscription_data_t connectData;
39
```

```
40      connectData.subscribe = htonl(UNSUBSCRIBE);
41      ret = sendto(sockfd, &connectData, sizeof(struct subscription_data_t),
42       0, (struct sockaddr*)&server, sizeof(struct sockaddr_in));
43      if (ret == -1) {
44       perror("sendto - unsubscribe");
45       return -1;
46      }
47      sockfd = 0;
48
49      return 0;
50     }
51
52    return -1;
53   }
54
55   void* recv_event()
56   {
57    int ret;
58    struct kcm_receive_event *msg;
59    msg = (struct kcm_receive_event*)malloc(getDataSize());
60
61    ret = recvfrom(sockfd, msg, sizeof(struct kcm_receive_event), 0, NULL, 0);
62    if (ret < 0){
63     perror("recvfrom - recv_event");
64     return NULL;
65    }
66
67    return (void*)msg;
68   }
```

```
69
70 int getValue(const void* data)
71 {
72  struct kcm_receive_event *msg = (struct kcm_receive_event*)data;
73
74  return ntohl(msg->value);
75 }
76
77 struct timeval getTimestamp(const void* data)
78 {
79  struct kcm_receive_event *msg = (struct kcm_receive_event*)data;
80  struct timeval timestamp;
81  timestamp.tv_sec = ntohl(msg->timestamp.tv_sec);
82  timestamp.tv_usec = ntohl(msg->timestamp.tv_usec);
83
84  return timestamp;
85 }
86
87 int getDataSize()
88 {
89  return sizeof(struct kcm_receive_event);
90 }
```

C.5 Adaptation Layer

The following sample code is an example of how an adaptation layer can be implemented. The adaptation layer is used in Chapter 6. It uses the ALCM API (see Appendix C.4) and shows how to use the different available functionality. The steps showed (implemented) are as follows:

- Subscribe
- Wait and receive a message (event)
- Print the contents of the received event; value, timestamp and size of event
- Repeat the previous two steps until a SIGINT signal is received
- When a SIGINT signal is received: Unsubscribe

More information is available in Section 5.3.4.

C.5.1 al.c

```
1  #include "alcm.h"
2  #include <signal.h>
3
4  #define IPADDRESS "127.0.0.1" //IP address of the server/kernel module
5
6  void die(int signal)
7  {
8   //Attempt to send unsubscription event to host subscribed to
9   if (unsubscribe() != 0) {
10   printf("ERROR: Unsubscribe failed\n");
11   exit(-1);
12  }
13
14  exit(0);
15 }
16
17 int main()
18 {
```

```
19  void* buffer;  //Event information buffer
20
21  signal(SIGINT, die);
22
23  //Attempt to send subscription request to defined IP address using
24  //the default port defined in alcm.h
25  if (subscribe(IPADDRESS, PORT) != 0) {
26   printf("ERROR: Failed to subscribe to %s:%d\n", IPADDRESS, PORT);
27   exit(-1);
28  }
29
30  while (1) {
31   buffer = recv_event();  //Blocking call to receive next event
32   if (buffer != NULL) {
33    printf("Received value %d at time %d:%d (%d bytes)\n",
34     getValue(buffer),   //Parse trigger value from event
35     (int)getTimestamp(buffer).tv_sec,  //Parse timestamp
36     (int)getTimestamp(buffer).tv_usec,
37     getDataSize());   //Get event data size
38
39     free(buffer);
40   }
41   else {
42    printf("ERROR: Received NULL message\n");
43    exit(-1);
44   }
45  }
46
47  return 0;
```

```
48 }
49
```

C.6 Pattern Generation Utility

The following output is a comparison between the pattern generation utility with and without trigger patterns. See Section C.1 for a description of the output.

C.6.1 patt_gen.diff

```
16:21:49.000000000 +0100
@@ -22,20 +22,21 @@
 #define UNKNOWN_FILE            0
 #define PATTERN_FILE            1
 #define SCENARIO_FILE           2

 /* Pattern types. Used in pattern file header, field "Pattern type".  */
 #define UNDEFINED_TYPE          0
 #define BITERROR                1
 #define PACKET_LOSS             2
 #define DELAY                   3
 #define BANDWIDTH               4
+#define TRIGGER     5

 /* Describes how patterns will behave, e.g. random bit errors, packet losses
    during certain intervals etc.  */
 #define RANDOM                  1
 #define GILBERT                 2
 #define POSITION                3
```

```
 #define INTERVAL                4
 /* #define EXPONENTIAL   5 */
 /* #define BOUNDED_PARETO  6 */

@@ -56,22 +57,22 @@
 #define PLOT                    3



 /* Forward declaration */
 int32_t         ceildiv(int32_t num, int32_t denum);

 /*Temporary hack to fix -graphx*/
 int32_t         graphx = 0;

-static char    *ptype[5] =
-    { "", "biterror", "packet loss", "delay", "bandwidth" };
+static char    *ptype[6] =
+    { "", "biterror", "packet loss", "delay", "bandwidth", "trigger" };
 static char    *dmode[3] = { "unspecified", "data-driven", "time-driven" };

 /* Header structure for compressed pattern. Total size 16 bytes  */
 struct p_header {
     char            magic[6]; /* "KauNet". 6 bytes  */
     uint8_t         version; /* 1 byte  */
     uint8_t         type; /* 1 byte. This byte is divided into two
      * parts. The two most significant bits
      * indicate if the pattern was produced
      * in time-driven or data-driven mode.
```

```
@@ -450,21 +451,21 @@
         1 The header is valid
 */
 int32_t
 check_header(struct p_header * hdr)
 {
     char            type = hdr->type & TYPE_PART;

     return (strncmp("KauNet", hdr->magic, 6) == 0) && /* Check magic bytes. */
  (hdr->version == VERSION) && (hdr->type & MODE_PART) && /* 2 bits, either 01,\
10 or 11. */
  ((type == BITERROR) || (type == PACKET_LOSS) ||
-  (type == DELAY) || (type == BANDWIDTH));
+  (type == DELAY) || (type == BANDWIDTH) || (type == TRIGGER));
 }



 /*  Advances pattern read state in compressed data (i.e. pattern file).

     Parameters:
         prs
               A pointer to the pattern read state structure.
               prs->distance is set to the number of zero bits before a
@@ -719,34 +720,49 @@
      prexlbl = "x100000 ";
  }

  divisor = yrange;
```

```
  sprintf(yticmode, "divisor = %d\nset ytics auto\n", divisor);
  sprintf(plotmode,
   "set bars small\nplot \"%s\" using (int($1)/divisor):((int($1)-1)%%divisor+\
0.5):(0.5) with yerrorbars linetype 1 pointtype 0\n",
   plot_txt);
     } else if (((hdr->type & TYPE_PART) == BANDWIDTH)
-         || ((hdr->type & TYPE_PART) == DELAY)) {
+         || ((hdr->type & TYPE_PART) == DELAY)
+      || ((hdr->type & TYPE_PART) == TRIGGER)) {

  if (!extract_values(prs, plotfile, PLOT))
      return 0;

  xrange = hdr->patternsize;
  yrange = (((prs->max_value / 100) * 100) + 100);

- typelbl =
-     (hdr->type & TYPE_PART) == BANDWIDTH ? "Bandwidth" : "Delay";
+ switch (hdr->type & TYPE_PART) {
+  case BANDWIDTH:
+   typelbl = "Bandwidth";
+   ylbl = "Bandwidth (kbps)";
+   break;
+  case DELAY:
+   typelbl = "Delay";
+   ylbl = "Delay (MilliSeconds)";
+   break;
+  case TRIGGER:
```

```
+   typelbl = "Trigger";
+   ylbl = "Trigger (Events)";
+   break;
+ }
+/* typelbl =
+     (hdr->type & TYPE_PART) == BANDWIDTH ? "Bandwidth" : "Delay"; */
  xlbl = (hdr->type & TIME_DRIVEN) ? "MilliSeconds" : "Packets";
- ylbl =
+/* ylbl =
      (hdr->type & TYPE_PART) ==
-     BANDWIDTH ? "Bandwidth (kbps)" : "Delay (MilliSeconds)";
+     BANDWIDTH ? "Bandwidth (kbps)" : "Delay (MilliSeconds)";*/

  gplbl = xlbl;

  sprintf(yticmode, "set ytics auto\n"); /* Automatic scaling */
  sprintf(plotmode, "plot \"%s\" with steps\n", plot_txt);

     } else
  return 0;  /* Unknown pattern type. */

     fprintf(gp, "\n#  %s Plot \n#----------------------\n", typelbl);
@@ -862,22 +878,22 @@
 display_usage()
 {
     printf("\nUsage:\n\n"
     "patt_gen -ber|-pkt -rand <filename> <mode> <size> <random_seed> <BVAL> \n"
     "patt_gen -ber|-pkt -ge   <filename> <mode> <size> <random_seed> <good_BER>\
<bad_BER> <good_tran_prob> <bad_tran_prob> \n"
```

```
     "patt_gen -ber|-pkt -pos  <filename> <mode> <size> <positions> \n"
     "patt_gen -ber|-pkt -pos  <filename> <mode> <size> -f <infilename> \n"
     "patt_gen -ber|-pkt -pos  <filename> <mode> <size> -r <infilename> \n"
     "patt_gen -ber|-pkt -int  <filename> <mode> <size> <interval-list>\
[<start_value>]\n"
     "patt_gen -ber|-pkt -int  <filename> <mode> <size> -f <infilename>\
[<start_value>]\n"
-    "patt_gen -del|-bw -pos  <filename> <mode> <size> <position-values> \n"
-    "patt_gen -del|-bw -pos  <filename> <mode> <size> -f <infilename> \n\n"
+    "patt_gen -del|-bw|-trig -pos  <filename> <mode> <size> <position-values> \n"
+    "patt_gen -del|-bw|-trig -pos  <filename> <mode> <size> -f <infilename> \n\n"
     "The first switch for the command controls which type of pattern should be\
produced: \n    bit-error, packet loss, delay change or bandwidth change.\n"
     "The second switch controls how the pattern should be generated: \n\
(pseudo-)randomly, using the gilbert-elliot model, or by explicilty giving\
positions either directly or in a file.\n\n"
     "   <filename> The name of the output file. \n"
     "   <mode> Specifies the generation mode (data / time). \n"
     "          The mode specification can influence the interpretation of later\
parameters. \n"
     "   <size> Specifies the length of the generated pattern. \n"
     "          The unit of the size can be either kilobytes, packets or\
milliseconds dependent on the type of pattern generated.\n"
     "   <random Seed> Gives the seed to use for random number generation. \n"
     "          Set to 0 to use internal seeding (Requires internet connectivity)\n"
     "   <positions> A comma separated list of positions where errors/losses are\
wanted.\n"
@@ -1718,20 +1734,23 @@
  size_factor = BITS_KBYTE;
```

```
     } else if (strcmp("-pkt", argv[1]) == 0) {
  hdr.type = PACKET_LOSS;
  size_factor = 1;
     } else if (strcmp("-del", argv[1]) == 0) {
  hdr.type = DELAY;
  size_factor = 1;
     } else if (strcmp("-bw", argv[1]) == 0) {
  hdr.type = BANDWIDTH;
  size_factor = 1;
+ } else if (strcmp("-trig", argv[1]) == 0) {
+ hdr.type = TRIGGER;
+ size_factor = 1;
     } else if (strcmp("-info", argv[1]) == 0) {
         filetype = check_filetype(argv[2]);
         if (filetype == SCENARIO_FILE) {
      print_scn_info(argv[2]);
  } else if (filetype == PATTERN_FILE) {
      dump(argv[2], INFO);
  } else {
    fprintf(stderr, "File is not scenario file or pattern file. Aborting");
    exit(EXIT_FAILURE);
  }
@@ -1858,21 +1877,21 @@
   if (argc == 8)
       tmpsv = strtoul_ck(argv[7]); /* Check if option startvalue is present */
  total_ones = set_interval(argv[6], &cpatt, hdr.patternsize * size_factor,\
tmpsv);
      }
  } else {
```

```
      printf ("Requested patterntype is unsupported for this type of patterns\n");
      exit(EXIT_FAILURE);
  }
  finalize_cpattern(&cpatt);

-    } else if ((hdr.type & TYPE_PART) == DELAY || (hdr.type & TYPE_PART) ==\
BANDWIDTH) {
+    } else if ((hdr.type & TYPE_PART) == DELAY || (hdr.type & TYPE_PART) ==\
BANDWIDTH || (hdr.type & TYPE_PART) == TRIGGER) {
  init_cpattern(outfileptr, &hdr, &cpatt);

  if (patterntype == POSITION) {
      if (strcmp(argv[6], "-f") == 0) {
   filebuf = read_textfile(argv[7]);
   total_ones = set_values(filebuf, &cpatt, hdr.patternsize * size_factor);
      } else {
   total_ones = set_values(argv[6], &cpatt, hdr.patternsize * size_factor);
      }
  } else {
```

C.7 The Internet Protocol Firewall

IPFW is the application used to load KauNet patterns into the kernel. Only a few changes are made to enable loading of trigger patterns.

C.7.1 dummynet.c

```
@@ -319,21 +319,21 @@
 ipfw_list_pipes(void *data, uint nbytes, int ac, char *av[])
```

```
 {
  int rulenum;
  void *next = data;
  struct dn_pipe *p = (struct dn_pipe *) data;
  struct dn_flow_set *fs;
  struct dn_flow_queue *q;
  int l;
 #ifdef KAUNET
  int i;
- char *pattern_types[MAX_PATTERNS] = { "Bit error", "Packet loss", "Delay change", \
"Bandwidth change" };
+ char *pattern_types[MAX_PATTERNS] = { "Bit error", "Packet loss", "Delay change", \
"Bandwidth change", "Trigger" };
  char pattern_string[500];
  char tempstr[256];
 #endif /* KAUNET */

  if (ac > 0)
   rulenum = strtoul(*av++, NULL, 10);
  else
   rulenum = 0;
  for (; nbytes >= sizeof *p; p = (struct dn_pipe *)next) {
   double b = p->bandwidth;
@@ -520,21 +520,21 @@
   case TOK_APPEND:
    if (scenario_set)
     errx(EX_USAGE, "You can not append a pattern while working with scenarios.\n");
    patterns[current_pattern].append = 1;
    printf("Trying to append patterns.\n");
```

```
    /* Fallthrough */

   case TOK_PATTERN:
    if ((current_pattern+1 > MAX_PATTERNS) && !patterns[current_pattern].append)
-      errx(EX_USAGE, "You have exceeded the limit of four patterns/pipe.\n");
+      errx(EX_USAGE, "You have exceeded the limit of %d patterns/pipe.\n", \
MAX_PATTERNS);
    if (scenario_set)
     errx(EX_USAGE, "You can not load a scenario and a pattern at the same time.\n");

      NEED1("You must supply a pattern file.\n");

    strcpy(filename, abs_filename(av[0]));

    /* Read and parse pattern file */
    if ((buff = file_to_buff(filename)) == NULL) {
     errx(EX_DATAERR, "Failed to load pattern file, exiting\n");
@@ -1080,21 +1080,21 @@

  return hdr;
 }

 /* Sync necessary data between pipe and pattern */
 void
 sync_info(struct pattern *p, struct p_header *hdr, struct dn_pipe *pipe)
 {
  p->pattern_pipe_nr = pipe->pipe_nr;
  p->timedriven = (hdr->type & MODE_PART & TIME_DRIVEN) ? 1 : 0;
```

```
- p->pattern_type = ((hdr->type & TYPE_PART) - 1) & (PACKET_LOSS | BITERROR | DELAY \
| BANDWIDTH );
+ p->pattern_type = ((hdr->type & TYPE_PART) - 1) & (PACKET_LOSS | BITERROR | DELAY \
| BANDWIDTH | TRIGGER);
  p->defer_start = 1;
  pipe->pattern_size[p->pattern_type] = hdr->patternsize;
 }

 /* Returns the absolute path of a given filename */
 char*
 abs_filename(char *filename) {

  char *name, *end = NULL;

@@ -1200,21 +1200,21 @@
  p->pattern_segments = hdr->arraysize;

  return hdr;

 }

 void
 print_pattern_status(struct dn_pipe *pipe)
 {
  int i;
- char *names[MAX_PATTERNS] = { "bit error", "packet loss", "delay change", \
"bandwidth change"};
+ char *names[MAX_PATTERNS] = { "bit error", "packet loss", "delay change", \
"bandwidth change", "trigger"};
```

```
  for(i = 0; i < MAX_PATTERNS; i++) {
   if (pipe->patterns[i] != NULL) {
    if ((pipe->pipe_nr == pipe->patterns[i]->pattern_pipe_nr) && (pipe->patterns[i]->\
cast == NO_PATTERN))
     printf("Pipe %d: Loading %s pattern from file %s, for %s mode.\n", pipe->\
pipe_nr, names[pipe->patterns[i]->pattern_type], pipe->patterns[i]->pattern_file\
, pipe->patterns[i]->timedriven ? "time driven" : "data driven");
    else if (pipe->patterns[i]->cast != NO_PATTERN)
      printf("Pipe %d: Trying to cast pattern from %s to %s\n", pipe->pipe_nr, \
names[pipe->patterns[i]->pattern_type], names[pipe->patterns[i]->cast]);
    else
     printf("Pipe %d: Sharing %s pattern with pipe %d, for %s mode.\n", pipe->\
pipe_nr, names[pipe->patterns[i]->pattern_type], pipe->patterns[i]->pattern_pipe_nr, \
pipe->patterns[i]->timedriven ? "time driven" : "data driven");
    if (pipe->patterns[i]->timedriven && !pipe->bandwidth)
```