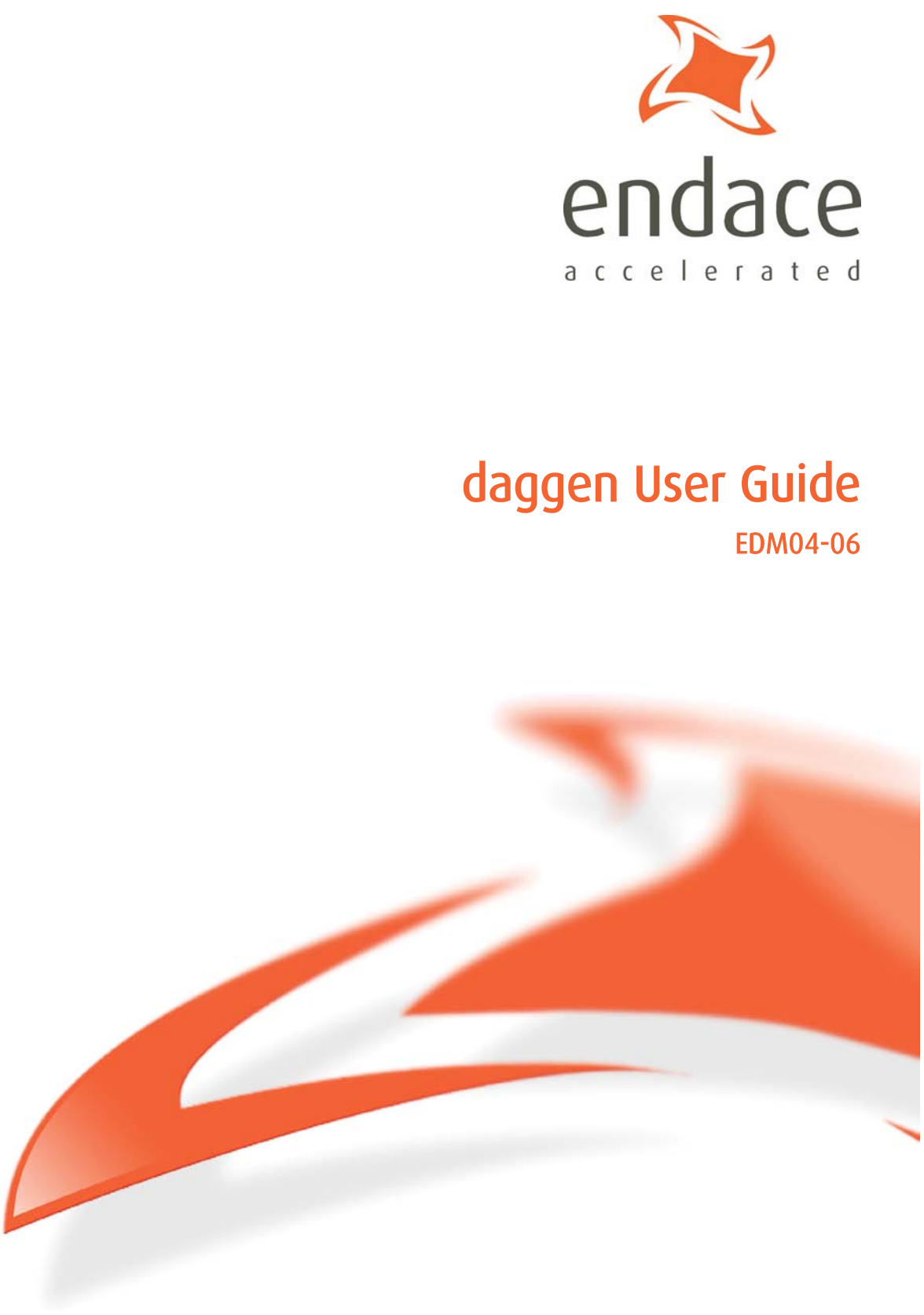

daggen User Guide

EDM04-06

Protection Against Harmful Interference

When present on equipment this manual pertains to, the statement "This device complies with part 15 of the FCC rules" specifies the equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the Federal Communications Commission [FCC] Rules.

These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment.

This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications.

Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at their own expense.

Extra Components and Materials

The product that this manual pertains to may include extra components and materials that are not essential to its basic operation, but are necessary to ensure compliance to the product standards required by the United States Federal Communications Commission, and the European EMC Directive. Modification or removal of these components and/or materials, is liable to cause non compliance to these standards, and in doing so invalidate the user's right to operate this equipment in a Class A industrial environment.

Disclaimer

Whilst every effort has been made to ensure accuracy, neither Endace Technology Limited nor any employee of the company, shall be liable on any ground whatsoever to any party in respect of decisions or actions they may make as a result of using this information.

Endace Technology Limited has taken great effort to verify the accuracy of this manual, but nothing herein should be construed as a warranty and Endace shall not be liable for technical or editorial errors or omissions contained herein.

In accordance with the Endace Technology Limited policy of continuing development, the information contained herein is subject to change without notice.

Website

http://www.endace.com

Copyright 2008 Endace Technology Ltd. All rights reserved.

Contents

Overview

daggen is a program which helps in creating traffic files. It can generate Ethernet, POS, ATM and AAL5 traffic files, in ERF and PCAP formats.

This User Guide describes the following components:

- daggen Configure File Format
- daggen Command Line
- daggen Config Example

daggen configure file format

Description

To use daggen a config file is required where the traffic is specified. Different kinds of packets are defined by the user who can also specify the order and amount of them.

Several random options are defined for the MAC address and payload of the packets, users can generate many traffic patterns with few lines.

The output file is a set of Endace Record Format [ERF] records 64-bit aligned containing the traffic specified before on a PCAP compatible file.

Simple config file example

Description

A simple config file generates 10 equal ethernet packets. Every packet has the:

- Source ethernet address 00:00:10:8a:9F:e2
- Destination ethernet address 01:6b:ca:13:24:bf
- 1500 bytes of payload.

The payload contents are random (dummy) bytes.

Example

The following is an example of a simple `daggen` config file.

```
// Our first test
packets {
    packet eth_802_3 first_packet  {
       src_addr ab:e1:c0:01:9F:e2;
       dst_addr c0:01:ab:e1:24:bf;
       payload dummy(1500);
    }
}
traffic {
    group traf_grp {
       send first_packet 10;
    }
}
```

The first line of the config file is a comment. Every character typed after the '`//`' and up to the end of the line is a comment and ignored. The behavior is the same as C++ or Java.

Every packet declaration is done inside the `packets` block. There must be only one `packets` block and must be placed at the beginning of the config file.

In this example it is declared an Ethernet 802.3 packet. Each packet declaration is done in the same way. First used is the keyword packet. This is followed by specifying the packet type, eth_802_3 in this case. The user then enters a packet identifier first_packet. The identifier is case sensitive and cannot begin with a number.

All `packet` parameters are specified between braces. The keywords '`src_addr`' and '`dst_addr`' refer to the Ethernet source and destination address.

The syntax of the MAC address is a 6 byte field expressed in hexadecimal notation with the bytes separated with a colon (:). The hexadecimal characters are case insensitive.

The packet is designated to have a payload size of 1500 bytes. Ethernet payload sizes range from 46 to 1500 bytes. The payload contents are random (dummy) bytes.

Every packet parameter ends with a semicolon (;).Once a packet is declared, the required traffic pattern is specified by the user. The commands to send the packets are grouped into a `group` statement. A user can define several traffic groups but only one of them is actually executed.

The above example has only one traffic group to be used, `traf_grp.` Traffic commands are specified inside the traffic group. The example shows the `send` command. This sentence writes the packet "`first_packet`" 10 times.

Random address

Description

More than one packet can be defined by a single packet declaration, by making the address random. Using the '*' character instead of a hexadecimal character makes the hexa-digit random.

As one hexadecimal character represents 4 bits, every '*' used randomizes an address in 16 values.

Given the following ethernet mac address:

```
00:10:ab:7e:5d:3*
```

With the '*' inserted, every packet within a 'send' command becomes different:

```
00:10:ab:7e:5d:30
00:10:ab:7e:5d:31
00:10:ab:7e:5d:32
[...]
00:10:ab:7e:5d:3F
```

The '*' can be inserted in different parts of a mac address. For example:

```
0*:**:f*:*a:b1:*6
```

results in an address like:

```
01:74:f3:9a:b1:46
05:4d:fe:1a:b1:66
07:5b:f6:5a:b1:76
03:31:f2:ba:b1:16
0a:ca:ff:ba:b1:d6
0d:75:f3:ea:b1:26
```

Numbers

Description

A `daggen` configuration file uses three ways to write numbers. Wherever a number is required any one of the following three formats can be used:

- In decimal format:

```
12345678
64
55793
```

- In hexadecimal format(prepending '`0x`'):

```
0x0123456789abcdef0123456789ABCDEF
0xaaAbda1
0xc001
```

- In binary format (prepending '`0b`'):

```
0b010010010
0b0001
0b10
```

Numbers are all considered unsigned.

Random numbers, patterns and distributions

Description

Packets can be generated with different payload sizes. `Daggen` syntax enables random size change, following a pattern or a statistic distribution.

The expressions used for changing payload size can also be used in other parts of the configuration file, so almost every number can be specified using these expressions.

Packets

The following are expression examples of packets generated by `daggen` with different payload sizes.

Example 1

Have a constant value:

```
x;
constant(x);
```

Example 2

Have random values between two numbers, both included:

```
uniform(x,y);
```

Example 3

Have distributed values on a normal distribution with center `'x'` and deviation `'y'`:

```
normal (x,y);
```

NOTE: At present the values of the normal distribution does not fit exactly with a truly normal distribution.

Example 4

Have values increasing between two numbers. When it arrives at the top value continue from the bottom(ROUND):

```
[x:y];
```

example:

```
payload dummy ([100:103]);
```

produces payload lengths:

```
100, 101, 102, 103, 100, 101, 102, 103, 100, ...
```

NOTE: if `y < x` values decrease.

Example 5

Have values increasing between two numbers. When it arrives at the top value continue decreasing, and after that increasing again (BOUNCE):

[x~y];

example:

```
payload dummy ( [100~103] );
```

produces payload lengths:

```
100, 101, 102, 103, 102, 101, 100, 101, ...
```

NOTE: if y < x values decrease.

Example 6

Have values ROUND or BOUNCE in steps greater than one:

```
[x:y] step z;
[x~y] step z;
```

example:

```
payload dummy ([100:106] step 2);
```

produces payload lengths:

```
100, 102, 104, 106, 100, 102,...
```

Example 7

Have values ROUND or BOUNCE in steps following a distribution:

```
[x:y] step uniform (a,b);
[x:y] step normal (a,b);
[x~y] step uniform (a,b);
[x~y] step normal (a,b);
```

Example 8

Have values ROUND or BOUNCE in steps ROUND or BOUNCE following a distribution:

```
[x:y] step [a~b] step normal (c,d);
```

Payloads

Description

daggen can produce different payload contents with some parts kept random and other parts following distributions.

Fixed payload

Description

The following line generates 1500 random bytes:

```
payload dummy (1500);
```

A fixed payload can be created for packets by writing a hexadecimal string with its contents. For example:

```
payload "0123456789abcdef0123456789abcdef";
```

The payload size is automatically calculated from the string length. But the string can have white spaces and carriage returns. The following payload is an example is from an IP-TCP-SSL packet captured with tcpdump and option -x:

```
payload  "4500 0066 058a 4000 4006 b1a9 c0a8 010d
c0a8 0101 d27a 03e1 64e4 bc07 158b 0862
8018 8218 8599 0000 0101 080a 04be 8fc0
0ad7 c5e5 1703 0000 2dbc b523 dbae 2e14
de4f 1bb7 a51b 3e92 3fbb 79b8 f049 e809
1a3c";
```

A captured IPv6, MPLS could also be used, or whatever other protocol and paste into the daggen configuration file.

Concatenation concept

Description

A packet payload can be constructed as a concatenation of two or more strings:

```
payload "aaaa" + "bbbb" + "cccc";
```

Using the three concepts of random payloads, strings and concatenation, a mixed payload can be produced: For example:

```
payload "aaaa" + dummy(2) + "cccc";
```

The sentence creates a payload sized 6 bytes with two random bytes in the middle. As many of these constructions can be used as are needed:

```
payload dummy(4) + "abcdef" + dummy([50:150] step 2)
+ "007" + "008" + dummy(normal(100,5));
```

NOTE: Odd sized strings are appended a trailing zero:

```
"abc" -> "abc0"
"007" + "008" -> "0070 0080"
```

Payload sizes cannot exceed 65535 bytes. Payload constructions exceeding the limit are truncated to 65535 bytes. Although payloads can be 64K big, not every link layer technology permits that size.

The final determination on payload size is in the link layer.

Counters

Description

Another item used in payload generation is counters. Counters are a distribution, random number or pattern stored in a variable. That variable can be used inside the payload.

Counters follow a distribution. Sizes can be 1, 2 or 4 bytes and are always unsigned. Declaration of counters is done inside the 'packets' group and before any packet declaration. A simple example is:

```
packets {
   counters {
          counter_2 [100:1000] size 2;
          counter_3 uniform(10,20) size 4;
          cnt_A [500~1000] step uniform (2,8) size 4;
   }

packet eth_802_3 packet_2 {
   payload counter_2 + counter_3(5) + cnt_A[3];
   }
}
```

Three counters are declared:

- One named '`counter_2`' which would have values from 100 to 1000, and again 100, which its size is 2 bytes.
- Another named '`counter_3`' taking random values between 10 and 20 with size 4 bytes.
- The last named '`cnt_A`' taking values between 500 and 1000 in random steps with sizes between 2 and 8 bytes. Its size is 4 bytes.

Counters are used in the payload sentence. In the above example the payload is constructed with these three counters.

- The first one is placed once at the beginning of the packet (2 bytes).
- The second is placed 5 times, with number in parenthesis, adding 4*5 bytes to the payload.
- The last one is placed only once, but not the entire counter (4 bytes) but only 3 bytes, with number in brackets.
- The final payload size is 25 bytes, being 2 + 4*5 + 3.

Example

In the following `daggen` example, two counters are used to change source and destination ports on a TCP packets.

```
spc uniform (1024, 65535) size 2;
dpc [6000:7000] size 2;
[...]
Payload  "4500 0066 058a 4000 4006 b1a9 c0a8 010d
         c0a8 0101"
         + spc + dpc +
         "64e4 bc07 158b 0862
         8018 8218 8599 0000 0101 080a 04be 8fc0
         0ad7 c5e5 1703 0000 2dbc b523 dbae 2e14
         de4f 1bb7 a51b 3e92 3fbb 79b8 f049 e809
         1a3c" + "0000 0000" + dummy([100~102]);
```


General Options

Description

Some general options can be set for the `daggen` config file. All of these options are defined in the block '`options`' at the beginning of the file.

Options

The following examples describe the general options for the `daggen` config file.

Output file

Define the output file for this test. Default value 'output.dag'

```
output_file <string>;
```

Verbose mode

Verbose mode. Shows extra messsages. Default value 'not verbose'.

```
verbose;
```

Random seed

Set a random seed. Exactly the same traffic pattern can be repeated several times. Without a `random_seed` statement different executions of `daggen` can generate different output files.

```
random_seed <number>;
```

Format output file

Choose format of output file between ERF or PCAP formats. In an ERF format, there can be more than one link layer packets. On PCAP format it is only possible to use one link layer format and it is automatically determined with the type of the first packet declared in the '`packets`' group.

```
output_format erf; (default)
output_format pcap;
```

Output interface

Choose which output interface to write the packets. This marks the ERF headers with the value where specified. Values permited are from 0 to 3. Other values will be truncated. The default value is 0 (interface 0). In PCAP format this field is ignored.

```
interface <number>;
```

Ethernet II traffic

Description

The only variation from ethernet II frame format to ethernet 802.3 is that Ethernet II frames have a 2 byte protocol field instead of a 2 byte length field.

Example

The following describes an Ethernet II packet declaration:

```
packet eth_II eth2_p {
   src_addr ab:e1:c0:01:**:**;
   dst_addr c0:01:ab:e1:**:**;
   protocol 0x0800;
   payload dummy(100);
}
```

POS traffic

Description

The generation of POS packets is as easy as generation of Ethernet packets.

Example

The following is an example of generating a POS packet which has all the fields that can be modified:

```
packet pos pos1 {
   address 0x0f;
   control 0x03;
   protocol_size 2;
   protocol 0x0800;
   payload uniform(100,9180);
   fcs_size 4;
}
```

Field definition

The POS traffic field definitions are described in the following table

Field	Definition
Address.	One byte. In PPP will take the fixed value 0xff (all stations). On cHDLC can be 0x0f (unicast) or 0x8f (multicast). Default value is 0xff.
Control.	One byte. In PPP will take the fixed value 0x03 (unnumbered info) and in cHDLC will be 0x00. Default value is 0x03.
Protocol size.	Can only take two values: 1 or 2 (bytes), which means the size of the protocol field in the POS header. Default is 2 bytes.
Protocol.	One or two bytes. The protocol in the payload of the POS packet. On cHDLC this field takes the same values as the protocol field in Ethernet header. Default value is 0x0000.
Payload.	Specifies POS payload contents.
FCS size.	Indicates how many bytes to use in the FCS (CRC) calculation for the packet. The values that can be taken are: 0, 2 or 4 bytes. In the first case there is no CRC calculation, nor appended at the packet. With 2 bytes FCS size it is used a CRC16 (CCITT) function and with FCS size of 4 bytes a CRC32 function will be used.

ATM Traffic

Description

With `daggen` ATM traffic based on single cells can be specified. The following example shows the syntax to construct ATM cells with `daggen`:

```
packet atm atm_1 {
   type uni;
   type nni;
   gfc 0x8;
   vpi 0x76;
   vci 0x5432;
   pt 0b0;
   clp 1;
   payload dummy(48);
   // hec auto calculated
}
```

Field definition

The ATM traffic field definitions are described in the following table

Field	Definition
Type uni and nni.	Can only have two values: uni or nni. UNI stands for 'User Network Interface' and NNI for 'Network to Network Interface'. The difference between them is that UNI has a Generic Flow Control field and NNI does not, allowing a 12-bit VPI field.
Generic Flow Control. (gfc)	4 bits. Only available in UNI ATM cells, it is ignored if entered in a NNI cell.
Virtual path identifier. (vpi)	8 bits sized in UNI ATM cells and 12 bits in NNI ATM cells. If the value exceeds the field size limit it will be truncated.
Virtual Channel Identifier. (vci)	16 bits
Payload Type. (pt)	3 bits.
Cell Loss Priority. (clp)	1 bit.
Header Error Control. (hec)	Is hardware automatically appended and does NOT exist in config file.
Payload.	48 bytes fixed size. Payloads smaller than 48 bytes will be zero padded.

The ATM cell format does not exist as a link layer in PCAP files, so they will be ignored and the output file not created.

AAL5 traffic

Description

Another type of traffic that can be defined with `daggen` is AAL5 traffic. AAL5 is not a link layer technology, but DAG cards support the transmit of this kind of frames doing the fragmentation by hardware.

DAG cards need not only the AAL5 frame but also some information about the ATM cells underlaying. For this reason it is also needed to specify the header of just one ATM cell for DAG cards to segment the AAL5 frame.

Syntax

The syntax for generating AAL5 with `daggen` is:

```
packet aal5 aal5_1 {
// ATM parameters
type uni;
gfc 0x8;
vpi 0x76;
vci 0x5432;
pt 0b0;
clp 1;

// AAL5 parameters
payload "aa aa 03 000000 0800" + dummy(16);
uu 0xDD;
cpi 0xEE;

// if CRC not specified or zero -> auto
crc 0xBBBBBBBB;
}
```

The first fields are common with ATM cells.

The payload field follows the same rules as other payloads as seen in the example. The first eight bytes specify that the AAL5 frame content is following the LLC/SNAP format containing an IP (0800) payload. Although any other payload can be used. The starting bytes make the frame visible through a PCAP based program, as a LLC/SNAP frame is expected.

Due to some hardware limitations, the maximum size allowed for the AAL5 payload is 65464 bytes. Automatic padding is added if necessary.

Field definition

The AAL5 payload field definitions are described in the following table

Field	Definition
User to User Identification. (uu)	One byte.
Common Part Indicator. (cpi)	One byte.
Length.	Field is automatically calculated and does not appear in the config file.
Cyclic Redundancy Check. (crc)	4 bytes. If not specified or specified with a value of zero,it is automatically calculated (CRC32).

Traffic commands

Description

There are two commands available to define complex traffic command patterns.

Traffic commands

The complex traffic commands are described in the following table.

Command	Description
send <packet_id> <how_many>	The send command writes <how_many> packets identified by <packet_id>. The parameter < how_many > can follow a distribution.
send <packet_id><how_many > snap <snaplength>	This is a variation of the send command allowing it to specify a snaplength for the packets written in this command. Snaplength can follow a distribution.
loop <how_many> { <commands> }	The loop command just iterates <how_many> times over a set of commands, which can be send or loop commands. As in the send command, <how_many> can follow a distribution, very different traffic patterns can be generated with few lines.

Traffic groups

Description

Traffic commands are grouped into traffic groups. Several traffic groups can be defined, but only one will be executed. That way the same config file can be used for many different tests by selecting the traffic group to execute. This selection is made through the command line.

```
traffic {
   group grpA {
          send packet 1;
   }

   group grpBBB {
          send packet 10;
   }
}
```

Description

Every general option can be overridden from command line arguments. Accordingly, the selection of the config file and the traffic group to be executed is specified with command line arguments.

Command line arguments

The command line arguments are described in the following table.

Command	Description
-e <fcs_size>	Select global FCS size. Select 0, 2 or 4 bytes. (default: chosen by packet).
-f <config_file>	Select the config file. (default: config.dag)
-i <interface>	Select output interface. (default: 0)
-o <output_file>	Select the output file. (default: output.dag)
-p	Select PCAP as output file format. (default: ERF format)
-r <random_seed>	Select the random seed. (default: time dependant)
-s <snaplength> (round down at 64-bit alignment) -S <snaplength> (round up at 64-bit alignment)	Truncate packets at some snaplength. (default: no truncation)
-v	Be verbose.(default: quiet)
-x <group_id>	Select traffic group to execute. (no default, if wrong reports error)

daggen Configuration File

Description

What can be done with a daggen configuration file is described in the following example.

Example

```
//////////////////////
// daggen script file //
//////////////////////


// General options
options {
   output_file "output3.erf";
   output_format erf;  // erf or pcap
   verbose;
   random_seed 5188;
   interface 2;
}

packets {
   counters {
          counter_1 20 size 1;
          counter_2 [100:1000] size 2;
          counter_3 uniform(0,150000) size 4;
          c4 uniform (0x2A00, 0x2D00) size 2;
          spc uniform (1024, 65535) size 2;
          dpc [6000:7000] size 2;
   }

   // Packet type A
   packet eth_802_3 packet_A {
          // when a sentence is specified more than once,
          // only the last one is used
          src_addr 00:08:71:**:c7:7*;
          dst_addr **:10:02:93:4A:**;
          payload dummy (1500);
          payload dummy ([46:1500] step [1:15] step uniform (2,4));
   }

   // Packet type B
   packet eth_802_3 packet_B {
          src_addr 0*:0*:0*:0*:0*:0*;
          dst_addr *0:*0:*0:*0:*0:*0;
          payload dummy (100);
       }
```

```
// Packet type B -DUPLICATE!
packet eth_802_3 packet_B {
       src_addr 00:FF:ff:00:00:01;
       dst_addr 00:FF:ff:00:00:02;
       payload dummy (100);
}

// Packet type A -DUPLICATE!
packet eth_802_3 packet_A {
       src_addr 00:08:71:B6:c7:7E;
       dst_addr 00:10:02:93:4A:1b;
       payload dummy (1500);
}

packet eth_802_3 packet_D { }

packet eth_802_3 packet_E {
       src_addr FF:FF:FF:FF:FF:**;
}

packet eth_802_3 F2004 {
       payload dummy(normal(1000,10));
}

packet pos pos1 {
       address 0x0f;
       control 0x03;
       protocol_size 2; // choose: 1 or 2 bytes
       protocol 0x0800; // IP
       payload dummy (uniform(100,9180));
       fcs_size 4; // choose: 0, 2 or 4 bytes
}

packet pos pos2 {

address 0x8f;
control 0;
protocol_size 1;
protocol 113;
payload dummy(100); // size of payload
fcs_size 2;
}

packet pos pos3 {
       fcs_size 0;
}
```



```
packet eth_II pII {
src_addr ab:e1:c0:01:**:**;
dst_addr c0:01:ab:e1:**:**;
protocol 0x0800;  // IP protocol
payload      "4500 0066 058a 4000 4006 b1a9 c0a8 010d
             c0a8 0101" + spc + dpc +
             "64e4 bc07 158b 0862
             8018 8218 8599 0000 0101 080a 04be 8fc0
             0ad7 c5e5 1703 0000 2dbc b523 dbae 2e14
                de4f 1bb7 a51b 3e92 3fbb 79b8 f049 e809
                1a3c" + "0000 0000" + dummy([100~102]);
}

packet eth_802_3 packet_z {
       src_addr 00:01:80:**:**:**;
       dst_addr 00:09:AB:**:CA:BA;
       payload "Abe1" + c4(7) + "f001 beef bad bed
       feed daf0e ca7 fa15e" + dummy (200);
}

packet atm atm_1 {
       type uni;
       gfc 0x8;
       vpi 0x76;
       vci 0x5432;
       pt 0b0;
       clp 1;

       payload dummy(48);
}

packet aal5 aal5_2 {
       // ATM parameters
       type uni;

       gfc 0x8;
       vpi 0x76;
       vci 0x5432;
       pt 0b0;
       clp 1;

       // AAL5 parameters
    Payload   "aa aa 03 000000 0800" +
              "6500 0066 058a 4000 4006 b1a9 c0a8 010d
              c0a8 0101 0123 0321" +
              "64e4 bc07 158b 0862
              8018 8218 8599 0000 0101 080a 04be 8fc0
              0ad7 c5e5 1703 0000 2dbc b523 dbae 2e14
              de4f 1bb7 a51b 3e92 3fbb 79b8 f049 e809
              1a3c" + "0000 0000" + dummy([100~102])
              + dummy(16);
uu 0xDD;
cpi 0xEE;
}
```

```
            traffic {

               group traffic_group_A {
                      send packet_A 1;

                      loop 4 {
                             send packet_A 100;
                             send packet_D 10;
                             send packet_B uniform(100,200);
                      send packet_E 2;
               }

               loop normal (100,8) {
                      loop [10:20] {
                      send F2004 [9~15] step 2;

               }

               send pos1    1;
               send pos2    1;
               send pos3    1;

            }

        }
        group traffic_group_B {
            loop [50:150] step uniform (2,8) {
               send packet_z 333;
               send pII     2002;
            }

            send packet_A 0xabc;

            loop 0x40 {
               send atm_1 0b0101110;
               send aal5_2 [100~0x100] step 0b101;
               }
        }

}
```


Version History

Version	Date	Reason
1		
2	August 2005	
3	September 2007	New template and correction to config section.
4	June 2008	Updated copyright information
5	November 2008	Updated front matter. Corrected simple config file example. Monor reformatting.