Sipat Triukose
Case Western Reserve University
revision: 042505

Wide Area Network Emulator HOWTO

1. Introduction

Wide area networks are complex, and require extensive testing before standardizations or deployment. It might not enough to test the protocol or system with mathematical method or simulation. Also, to test the protocol in a real wide area network is not always easy to do, especially, if the topology has to be changed frequently.

The idea of WAN emulator, as shown in Figure 1, is to implement a system that emulates characteristics and behaviors of wide area network such as delay, loss rate, bandwidth limitation, queue algorithm, randomness, and so forth. Also, it is easy enough to control the characteristics or topology.

There are several types of emulators ranging from dedicated hardware, commercial software, or open source distribution. This document presents an open source emulator on PC based systems. For academic research purposes, this open source system offers many advantages such as increased accessibility in terms of cost and hardware, avoidance of legal problem, and its entirely open source nature facilitates the in-depth understanding of the system and leaves opportunities to further develop it in the future.

Wide Area Network Emulator

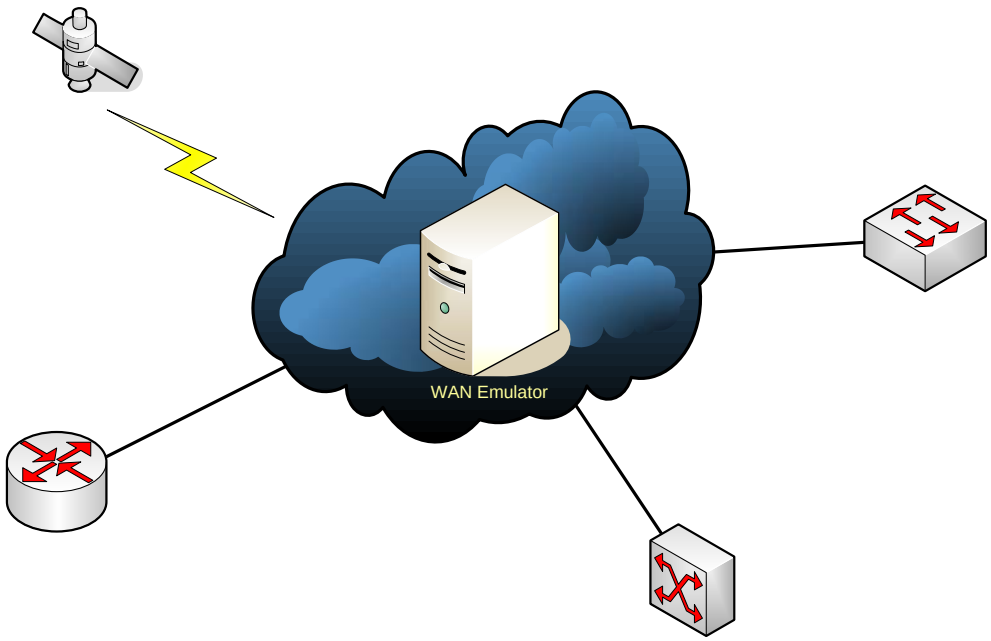

Figure 1

2. Overview

This section overviews some important concepts necessary for building and using the WAN emulator. Apart from the basic FreeBSD usage, the knowledge of sysctl's variables and IP firewalls is very helpful in understanding and manipulating the WAN emulator system, introduced later in this document.

The sysctl's variables control many FreeBSD system behaviors such as firewall, dummynet, and bridging. The meaning and control of each variable is very important in customizing the system.

IP firewall is also very important in term of networking and security. By matching the rule set, the firewall manipulates packets, passed through the system, by allowing, denying, or directing the packet, for example. In case of the emulator, the firewall directs certain flow to certain Dummynet pipes, which behave as wide-area links. Also, the WAN topology is emulated by manipulating the order in which firewall directs the packets to multiple dummynet's pipes. More elaborated examples will be covered in the "Using WAN Emulator" section below.

2.1 Sysctl variables

The FreeBSD kernel states are described using a "Management Information Base" or MIB style name, presented in a dotted set of components. The **sysctl** is the utility that allow privilege users to set or retrieves the kernel state variable. The following are the related Sysctl's variables that control the behaviors of firewall, bridging, and dummynet:

Controlling ipfw

The firewall is mostly controlled by ipfw, and the sysctl variables only serve to give global configuration and default parameters.

net.inet.ip.fw.enable: 1
: Enables firewall in the IP stack

net.inet.ip.fw.one_pass: 1
: Forces a single pass through the firewall. If set to 0, packets coming out of a pipe will be reinserted into the firewall starting with the rule after the matching one. NOTE: there is always one pass for bridged packets.

net.inet.ip.fw.dyn_buckets: 256 (read-only)
: Current hash table size used for dynamic rules.

net.inet.ip.fw.curr_dyn_buckets: 256
Desired hash table size used for dynamic rules.

net.inet.ip.fw.dyn_count: 3
Current number of dynamic rules. (read-only)

net.inet.ip.fw.dyn_max: 1000
Max number of dynamic rules. If you exceed this limit, you will have to wait for a rule to expire before being able to create a new one.

net.inet.ip.fw.dyn_ack_lifetime: 300
net.inet.ip.fw.dyn_syn_lifetime: 20
net.inet.ip.fw.dyn_fin_lifetime: 20
net.inet.ip.fw.dyn_rst_lifetime: 5
net.inet.ip.fw.dyn_short_lifetime: 5
Lifetime (in seconds) for various types of dynamic rules.

Controlling dummynet

Also dummynet is mostly controlled by ipfw, with the sysctl variables serving mostly for default parameters.

net.inet.ip.dummynet.hash_size: 64
Size of hash table for dynamic pipes.

net.inet.ip.dummynet.expire: 1
Delete dynamic pipes when they become empty.

net.inet.ip.dummynet.max_chain_len: 16
Max ratio between number of dynamic queues and hash buckets. When you exceed (max_chain_len*buckets) queues on a pipe, packets not matching any of these will be all put into the same default queue.

Controlling bridging

Bridging is almost exclusively controlled by sysctl variables.

net.link.ether.bridge_cfg: ed2:1,rl0:1,
set of interfaces for which bridging is enabled, and cluster they belong to.

net.link.ether.bridge: 0
Enable bridging.

net.link.ether.bridge_ipfw: 0
Enable ipfw for bridging.

2.2 IPFW and Firewall

The ipfw utility is the user interface for controlling the firewall and the dummynet traffic shaper in FreeBSD. In this WAN emulator, incoming packet will be injected into the firewall and then the firewall plays a role in deciding how to forward each packet and how each packet interacts with dummynet. Therefore, it is very helpful to understand how the packet moves in the firewall stack.

A packet is checked against the active rule set in multiple places in the protocol stack, under the control of several sysctl variables. These places and variables are shown below, and it is important to have this picture in mind in order to design a correct rule set.

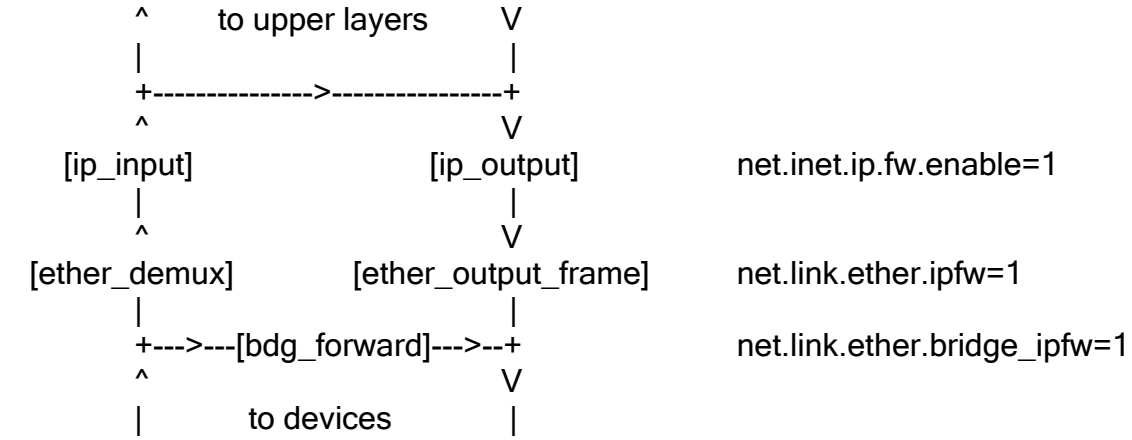

Figure 2

As can be noted from the above picture, the number of times the same packet goes through the firewall can vary between 0 and 4 depending on packet source and destination, and system configuration.

Note that as packets flow through the stack, headers can be stripped or added to it, and so they may or may not be available for inspection. E.g., incoming packets will include the MAC header when ipfw is invoked from **ether_demux()**, but the same packets will have the MAC header stripped off when ipfw is invoked from **ip_input()**.

Further explanations will be given as needed in the rest of this document. Moreover, Ipfw has many features that are beyond the scope of this document. For more information about ipfw, please consult ipfw manpage.

3. Implementing Bridge and Dummynet

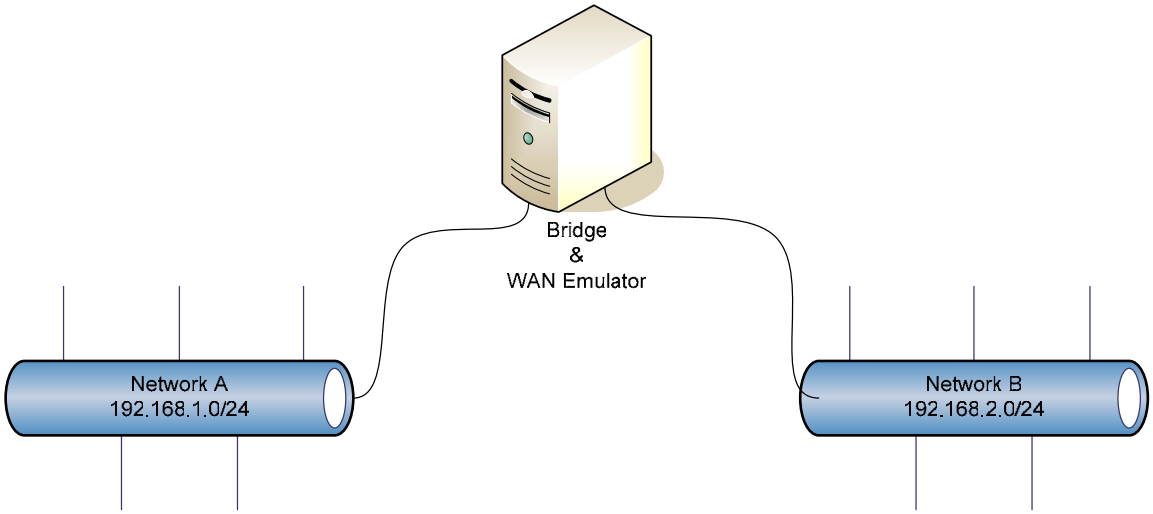

Figure 3

Assuming prior knowledge of basic FreeBSD usage, the previous section provided adequate background to start building a WAN emulator. The sample system, as shown in Figure 3, will be used to demonstrate how to implement the WAN emulator.

In brief, the WAN emulator is a FreeBSD box functioning as a bridge connecting multiple networks together. Then, the dummynet module will be used to emulate WAN characteristics such as delay, loss rate, bandwidth, and queue policy. The topology of WAN is specified by the firewall rule set.

The hardware requirement for the emulator is an IBM compatible PC with at least two network interfaces and FreeBSD installed. The implementation on this document uses FreeBSD 5.3.

The implementation could be categorized to four following steps:

- Install and configure the FreeBSD system
  This document assumes as a prerequisite basic installation and configuration of FreeBSD. More information concerning how to setup FreeBSD can be found at www.freebsd.org

- Reconfigure and recompile kernel to support Bridge and Dummynet
  The necessary kernel options required for the WAN emulator will be discussed in “Kernel Options” subsection. This document assumes prior knowledge of how to configure and compile the FreeBSD kernel.

- Configure the FreeBSD box to function as a bridge
  This step will be discussed in the “Enable Bridge” subsection.

- Assigns firewall rule set to emulate the Wide Area Network.
  The “Using the WAN Emulator” section is devoted for this step. The discussion will be given through a series of examples.

3.1 Kernel Options

Bridging and Dummynet are both running in the kernel space. Therefore, proper kernel options are necessary to enable both features.

To enable kernel support for bridging, add the following option:

- options BRIDGE

The following options in the kernel configuration file are related to dummynet operation:

IPFIREWALL	enable ipfirewall (required for **dummynet**)
IPFIREWALL_VERBOSE	enable firewall output
IPFIREWALL_VERBOSE_LIMIT	limit firewall output
DUMMYNET	enable **dummynet** operation
NMBCLUSTERS	set the amount of network packet buffers
HZ	set the timer granularity

In general, the following options are required to enable dummynet:

- options IPFIREWALL
- options DUMMYNET
- options HZ=10000

In addition, the number of mbuf clusters where network packets are stored could be increased according to the sum of the bandwidth-delay products and queue sizes of all configured pipes.

3.2 Enabling the Bridge

As enabled in the kernel space, the bridge can be enabled or disabled at runtime via sysctl interface.

Add this line:

net.link.ether.bridge.enable = 1 (set to be 0 to disable the bridge)

to /etc/sysctl.conf to enable the bridge at runtime,

net.link.ether.bridge.config = if_1, if_2, …, if_n

to specify the interfaces on the bridge (if_x represent network interface), and

```
net.link.ether.bridge.ipfw = 1
net.link.ether.ipfw=1
net.inet.ip.fw.enable=1
```

As shown in Figure 2, to allow the packet pass through the firewall bridge, Ethernet, and IP stack respectively. The dummynet emulator could be deployed in multiple layers; however, in this document, it will be mainly described in the IP stack.

At this point, the system is fully operational. Let run a simple test on bridge function here.

The very simple Bridge without any restrictions could operate by only single rule:

```
$ ipfw -f f
$ ipfw add 65500 allow ip from any to any
```

This bridge will forward traffics on any networks to any networks.

To do the test, please connect one Linux machines to the network 192.168.1.0/24 and another one to 192.168.2.0/24. Then, set the interface and routing as follow:

Linux Box A (in network 192.168.1.0/24):

```
$ ipconfig eth0 192.168.1.10 netmask 255.255.255.0 broadcast 192.168.1.255 up
$ route add -net 192.168.2.0/24 eth0
```

Linux Box B:

```
$ ipconfig eth0 192.168.2.10 netmask 255.255.255.0 broadcast 192.168.2.255 up
$ route add -net 192.168.1.0/24 eth0
```

Then, Linux Box A and B should be able to ping IP of each other.

Alternative to Bridging

The WAN emulator in this document is implemented as a bridge. However, the emulator could be implemented to operate as a router as well. In brief, this could be done by enabling the FreeBSD box to forward IP packets (enable IPFORWARD) and set the default route of a computer in each network to be the IP address of the emulator network interface that is physically connected to that network. The discussion of advantages and disadvantages of Bridging versus Routing is beyond the scope of this document.

4. Using WAN Emulator

The emulator is a combination of different services. Traffic is transferred between networks via bridging. The heart of emulator is the concept of a virtual link, called pipe. With the help of dummynet, the pipe can introduce behaviors like delay, loss rate, and bandwidth. Firewall rule set help redirect any particular traffic to a certain pipe to form simple or even complex WAN topologies.

4.1 Simple WAN Emulator

In the first example, packets will be forced to go through the firewall only once. Therefore, the **one_pass** option needs to be enabled.

```
$ sysctl net.inet.ip.fw.one_pass = 1
```

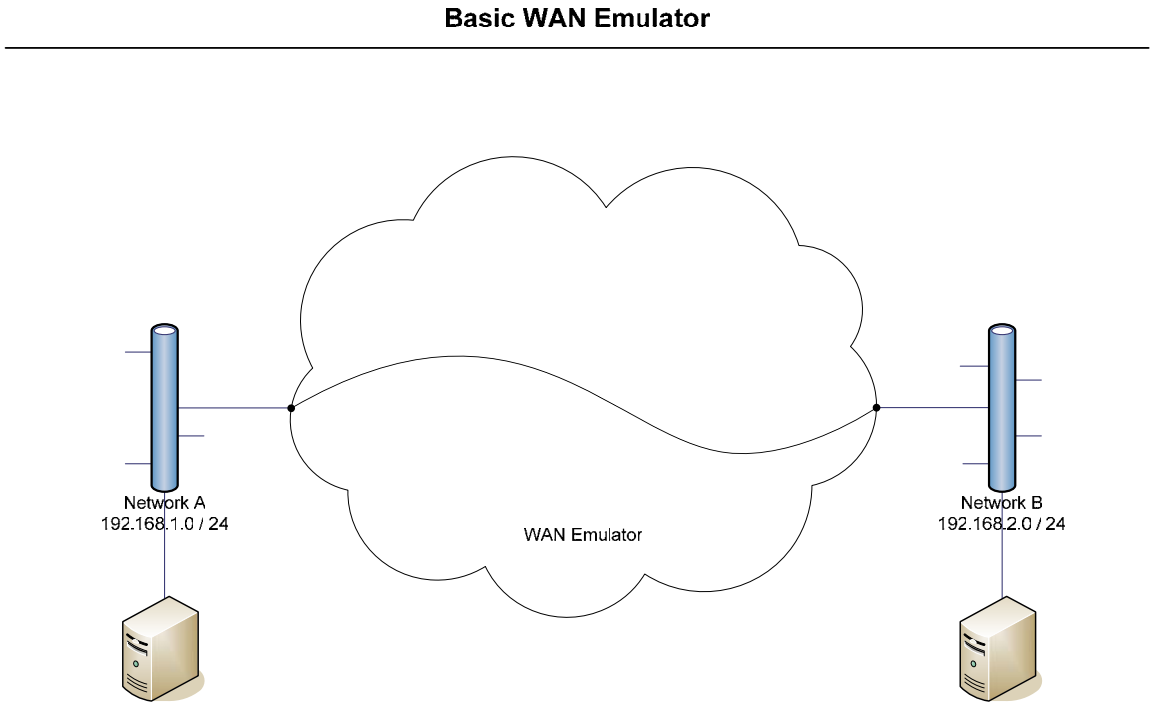

Figure 4

The basic full-duplex channels, as shown above in the figure 4, could be formed by the following rules:

```
NETA=192.168.1.0/24
NETB=192.168.2.0/24

# Clear all the rules and pipes
ipfw -f
ipfw pipe -f
```

```
# Add necessary rules and pipes
ipfw add 100 allow layer2 not mac-type ip
ipfw add 1000  pipe 1 ip from $NETA to $NETB
ipfw add 1010  pipe 2 ip from $NETB to $NETA
ipfw add 65000 deny ip from any to any

# Configure the pipes. Replace X, Y, and Z with preferred numbers.
ipfw pipe 1 config delay X ms plr Y bw Z kb
ipfw pipe 2 config delay X ms plr Y bw Z kb
```

Note: Make sure that sysctl's variable **net.inet.ip.fw.one_pass = 1**

The first two commands clear up all the rules and pipes. The third rules let all the non-ip packets, such as ARP packets, pass though. The forth and fifth rules tell the firewall to redirect traffic that matches the condition to the corresponding pipe. Traffic from NETA to NETB will be forwarded to pipe 1 and traffic from NETB to NETA to pipe 2. The 6th rule is the default rule, which is optional because the system has the default rule number 65535 (this default can be changed in the kernel configuration). The later example will assume this default rule, "65535 deny all from any to any."

In the pipes configuration, the **delay** specifies the delay of the pipe in milliseconds. The **plr** parameter is a packet loss rate value ranging from 0 to 1. The **bw** parameter is the bandwidth of the pipe. The default 0 will be assumed for plr and bw if not specified in the configuration statement. The value 0 means ideal channel and unlimited bandwidth for **plr** and **bw** (Of course, limited by physical bandwidth of the network interface) respectively.

By controlling these parameters, various kind of scenarios can be emulated. Lossy channel could be built by setting the **plr** parameter. The ADSL link could be emulated by setting different bandwidth values for the uplink and downlink directions. A long delay link, such as one to a satellite or to the moon, could be emulated by setting the delay parameter.

Jitter in Delay

In the previous example, the link introduces a constant delay. However, in reality, the delay is not always a constant. The following example shows how to add delay jitter.

This example uses the firewall **prob** rule as a key to add an uncertainty in delay. The concept of **prob**, quoted from ipfw manpage, is described below:

```
prob match_probability
     A match is only declared with the specified probability (floating
     point number between 0 and 1).  This can be useful for a number
     of applications such as random packet drop or (in conjunction
     with dummynet(4)) to simulate the effect of multiple paths leading
     to out-of-order packet delivery.
```

The script of simple WAN emulator with delay jitter added:

```
# Note that the one_pass is enabled
NETA=192.168.1.0/24
NETB=192.168.2.0/24

# Clear all the rules and pipes
ipfw -f
ipfw pipe -f

# Add necessary rules and pipes
ipfw add 100 allow layer2 not mac-type ip

ipfw add 1000 prob .1   pipe 10 ip from $NETA to $NETB
ipfw add 1001 prob .25 pipe 11 ip from $NETA to $NETB
ipfw add 1002              pipe 12 ip from $NETA to $NETB

ipfw add 1010 prob .1   pipe 20 ip from $NETB to $NETA
ipfw add 1011 prob .25 pipe 21 ip from $NETB to $NETA
ipfw add 1012              pipe 22 ip from $NETB to $NETA

ipfw add 65000 deny ip from any to any

# Configure the pipes.
ipfw pipe 10 config delay 400ms
ipfw pipe 11 config delay 150ms
ipfw pipe 12 config delay 100ms

ipfw pipe 20 config delay 400ms
ipfw pipe 21 config delay 150ms
ipfw pipe 22 config delay 100ms
```

In the example above, since **one_pass** is enabled, only one out of the rule 1000 – 1002 is matched for the packets from network A to B, and similarly the packets from network B to A will match only one rule out of 1010 - 1012. In this example, mostly, 65%, the delay of the link is 100ms. However, 150ms and 400ms jitter is embedded with the probability of 25 and 10 percents respectively.

Multi Paths and Out-of-Order Packet delivery

In the real wide area network, different packets can be routed through different paths, which can have different delays. Therefore, packets can arrive out-of-order. Out-of-order arrivals can be emulated by means of the **prob** rule. The following is an example of such system:

```
# Note that the one_pass is enabled
NETA=192.168.1.0/24
NETB=192.168.2.0/24

# Clear all the rules and pipes
```

```
ipfw –f
ipfw pipe -f

# Add necessary rules and pipes
ipfw add 100 allow layer2 not mac-type ip

ipfw add 1000 prob .1   pipe 1 ip from $NETA to $NETB
ipfw add 1001 prob .2   pipe 2 ip from $NETA to $NETB
ipfw add 1002           pipe 3 ip from $NETA to $NETB

ipfw add 1010 prob .1   pipe 1 ip from $NETB to $NETA
ipfw add 1011 prob .2   pipe 2 ip from $NETB to $NETA
ipfw add 1012           pipe 3 ip from $NETB to $NETA

Ipfw add 65000 deny ip from any to any

# Configure the pipes.
ipfw pipe 1 config delay 800ms
ipfw pipe 2 config delay 250ms
ipfw pipe 3 config delay 100ms
```

This example shows a 3 route network, and probabilities of 10, 20, and 70 percents that packets will be routed to pipe 1, 2, and 3 respectively. With different delays on each route, out-of-order delivery is possible.
HERE.

4.2 Complex Topology

As explained in the previous subsection, the WAN emulator introduces delays, losses, and bandwidth limitations through the parameters in each logical link (pipe). This section will introduce how to emulate a more complex network topology formed as the interconnection multiple links.

The idea is to let each packet go through multiple firewall rules before leaving the emulator. The particular network topology can be formed by manipulating the sequence of rules that packets in each flow go through. Please note that "ipfw one_pass" needs to be disabled by setting the following sysctl's variable:

```
$ sysctl net.inet.ip.fw.one_pass = 0
```

Otherwise, packets would not go through multiple pipes.

To manipulate the sequence of rules that packet will go through, the firewall rule actions are the key. The details of the following actions are necessary to understand how to construct a multiple links topology.

Related Rule Actions (Excerpted from ipfw manual page)

allow | accept | pass | permit
Allow packets that match rule. The search terminates.

deny | drop
Discard packets that match this rule. The search terminates.

pipe *pipe_nr*
Pass packet to a dummynet's pipe (for bandwidth limitation, delay, etc). The search terminates; however, on exit from the pipe and if the *sysctl(8)* variable **net.inet.ip.fw.one_pass** is not set, the packet is passed again to the firewall code starting from the next rule.

skipto *number*
Skip all subsequent rules numbered less than *number*. The search continues with the first rule numbered *number* or higher.

Multiple Links Topology

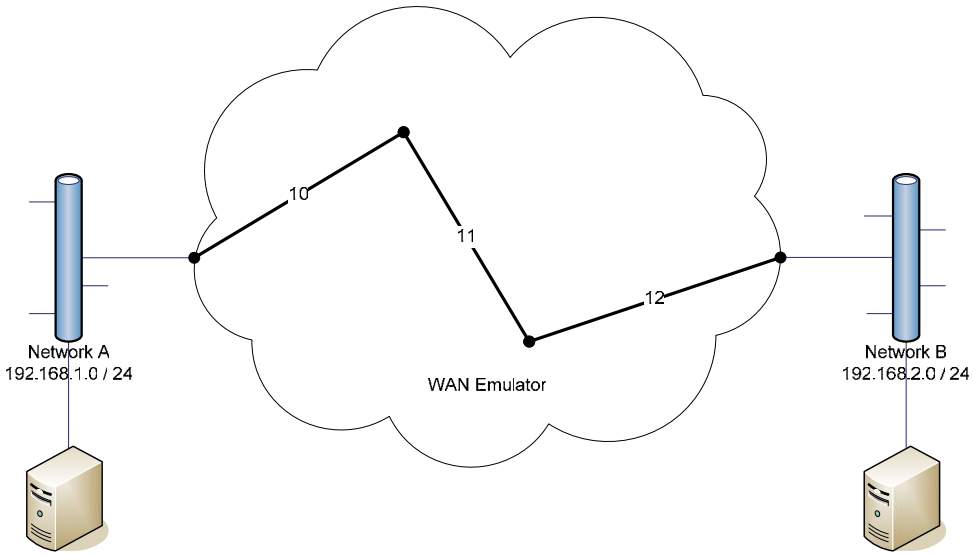

Figure 5

Each logical link is represented by a dummynet pipe, which could be configured independently. Therefore, the emulator could emulate interconnected links with different characteristic on each link. How the logical links interconnect is correlated to the order of pipe rules. With the one_pass disabled, the packet can be forced to pass through series of pipes, where the order of pipe rules represents the logical links' topology. In the case of one_pass disabled, the search is terminated by either **allow** of **deny** rules. The **skipto** action is used to direct each flow to a proper termination.

The following script will emulate a three hop network as shown in the figure 5.

```
NETA=192.168.1.0/24
NETB=192.168.2.0/24

# Clear all the rules and pipes
Ipfw –f
Ipfw pipe -f

# Add necessary rules and pipes
Ipfw add 100 allow layer2 not mac-type ip

# Route from NETA to NETB
ipfw add 1000  pipe 10 ip from $NETA to $NETB
ipfw add 1001  pipe 11 ip from $NETA to $NETB
ipfw add 1002  pipe 12 ip from $NETA to $NETB
ipfw add 1009  skipto 65450 ip from $NETA to $NETB

# Route from NETB to NETA
ipfw add 1010  pipe 12 ip from $NETB to $NETA
ipfw add 1011  pipe 11 ip from $NETB to $NETA
ipfw add 1012  pipe 10 ip from $NETB to $NETA
ipfw add 1019  skipto 65450 ip from $NETB to $NETA

ipfw add 56400 skipto 65530 ip from any to any
ipfw add 65500 allow ip from any to any
# assume the default rule, 65535, is deny all traffic

# Configure the pipes
Ipfw pipe 10 config delay 0 ms
Ipfw pipe 11 config delay 0 ms
Ipfw pipe 12 config delay 0 ms
```

The first two ipfw commands will flush all the rules and pipe configurations. The flow from 192.168.1.0/24 to 192.168.2.0/24 will go through rules 1000 – 1002, afterward, the packet will be skipped to just before the rule 65500 by the **skipto** action of the rule 1009. Finally, the flow will be terminated by the rule 65500, which allows the packet to pass to network B, 192.168.2.0/24. In the reverse direction, the backward flow from network B to A will go through rules 1010 – 1012. Then will be skipped to the 65500 rule which will terminate the search. Other IP traffic will match the rule 56400 and then will be skipped to the default rule, 65535, which is a **deny** action in this example.

Now, different kind of scenarios can be configured by assigning different values to variables, such as delay, bandwidth, queue policy, or loss rate, for each pipe.

The following will present another example of a more complex topology in which the forward and backward flow use different routes.

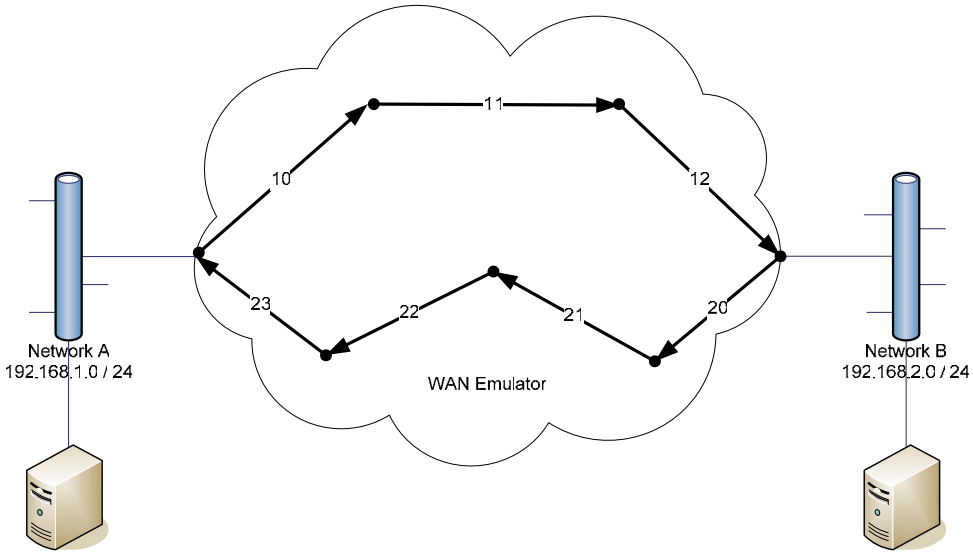

Figure 6

The network topology as shown in figure 6 can be formed by the following script.

```
NETA=192.168.1.0/24
NETB=192.168.2.0/24

# Clear all the rules and pipes
Ipfw -f
Ipfw pipe -f

# Add necessary rules and pipes
Ipfw add 100 allow layer2 not mac-type ip

# Route from NETA to NETB
ipfw add 1000  pipe 10 ip from $NETA to $NETB
ipfw add 1001  pipe 11 ip from $NETA to $NETB
ipfw add 1002  pipe 12 ip from $NETA to $NETB
ipfw add 1009  skipto 65450 ip from $NETA to $NETB

# Route from NETB to NETA
ipfw add 1010  pipe 23 ip from $NETB to $NETA
ipfw add 1011  pipe 22 ip from $NETB to $NETA
ipfw add 1012  pipe 21 ip from $NETB to $NETA
ipfw add 1013  pipe 20 ip from $NETB to $NETA
ipfw add 1019  skipto 65450 ip from $NETB to $NETA
```

```
ipfw add 56400 skipto 65530 ip from any to any
ipfw add 65500 allow ip from any to any
# assume the default rule, 65535, is deny all traffic

# Configure the pipes
Ipfw pipe 10 config delay 0 ms
Ipfw pipe 11 config delay 0 ms
Ipfw pipe 12 config delay 0 ms

Ipfw pipe 20 config delay 0 ms
Ipfw pipe 21 config delay 0 ms
Ipfw pipe 22 config delay 0 ms
Ipfw pipe 23 config delay 0 ms
```

The first two ipfw command will flush all the rules and pipe configurations. The difference from previous example is that the forward and backward flow will match different set of pipes. The flow from network A to B will go through rules 1000 – 1002, which will be passed to pipe 10, 11, and 12 respectively. On the other hand, the flow from network B to A will go through rules 1010 – 1013, which will be passed to pipe 20, 21, 22, and 23 respectively. Both the flow from network A to B and from network B to A will be directed by the **skipto** action to the rule 65500, which will allow the traffic to pass through the emulator and terminate the search in the firewall rule set. Other IP's traffic will be directed to the default rule, which is a **deny** action in this example.

Link Failure

Interestingly, the link could be brought down by the following script:

```
ipfw del 1011
ipfw add 1011 deny ip from $NETB to $NETA
```

The above brings the link 22 down. If the computer in network A, called computer A, pings the computer in network B, called computer B, the **tcpdump** on the interface of computer A will show only "echo request." On the other hand, the **tcpdump** on the interface of computer B will show both "echo request" and "echo reply." This show that the forward and backward traffic between computers A and B use different routes and that link 22 is in fact brought down.

5. Comments

5.1 TCP Effect

A long round trip time (large delay on some pipe) might cause bandwidth underutilization due to the TCP window size. For example, in case of the ideal channel with 1 MByte/s link and 10 second RTT, should the TCP window size is 30000 Byte, the bandwidth will be utilized at only around

Window size / RTT

which will be 3000 Byte/s in this example.

5.2 Error Rate Simulation

In a real scenario, the delivered packet might contain some bit error. The dummynet currently does not support this feature. The author would like to encourage the development of this feature. It could be developed as a queue policy that can modify certain a number of bits with certain probability. Then, it could be added to any pipe to simulate the error in packet.

5.3 Queue Algorithm

Although not discussed explicitly in this document, dummynet could also introduce different kind of queuing algorithms for each pipe such as red, gred, or even custom implemented algorithm. For further information, please consult IPFW(8) and DUMMYNET(4) manpages.

6. Acknowledge

We thank Dr. Vincenzo Liberatore, Case Western Reserve University, for guiding and supporting, Luigi Rizzo for the very useful dummynet module, Ewan Markensohn for the document that helped me get start, Akkradech Watcharapupong, KMITL University, Thailand, for your invaluable comments and suggestions.

7. Feedback

Comments and suggestions will be invaluable. Please contact Mr. Sipat Triukose at sxt85@case.edu

Related Resources

Dummynet:

- Dummynet: a simple approach to the evaluation of network protocols, ACM Computer Communication Review, Jan. 1997
  http://portal.acm.org/citation.cfm?id=251012
- dummynet document by Luigi Rizzo, the creator of dummynet
  http://info.iet.unipi.it/~luigi/ip_dummynet
- FreeBSD dummynet manual page, DUMMYNET(4)
- Dummynet articles and tutorials
  - http://www.bsdnews.org/02/dummynet.php
  - http://ai3.asti.dost.gov.ph/sat/dummynet.html
  - http://cs.ecs.baylor.edu/~donahoo/tools/dummy/

FreeBSD:

- FreeBSD Handbook, by The FreeBSD Document Project
  http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/index.html
- FreeBSD Installation Guide
  http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/install.html
- FreeBSD Bridge manual page, BRIDGE(4)
- FreeBSD ipfw manual page, IPFW(8)