Netlag: A kernel module for simulating geographically large networks.

Erin Johnson

The over all goal of this project was to create a kernel module that would allow the simulation of geographically large networks by delaying packets from a specified IP by a specified amount.

Main Goals:

1. Find a way to monitor IPv4 packets as they arrive
2. Remove packets from propagating to userspace
3. Implement hrtimers to create a delay
4. Re-inject packets and let them naturally propagate up to userspace

Sub Goals:

1. Proper memory management
2. Use pre-existing functions and structures
3. Allow user to set delay and IP at init
4. Allow user to set delay and IP after init
5. Implement handling for IPv6
6. Allow for multiple IPs to be delayed

All main goals and subgoals one through three were implemented. While four through six may seem trivial, much of the time for this project was spent attempting to understand what existed within the kernel.

The rest of this writeup is laid out as follows:

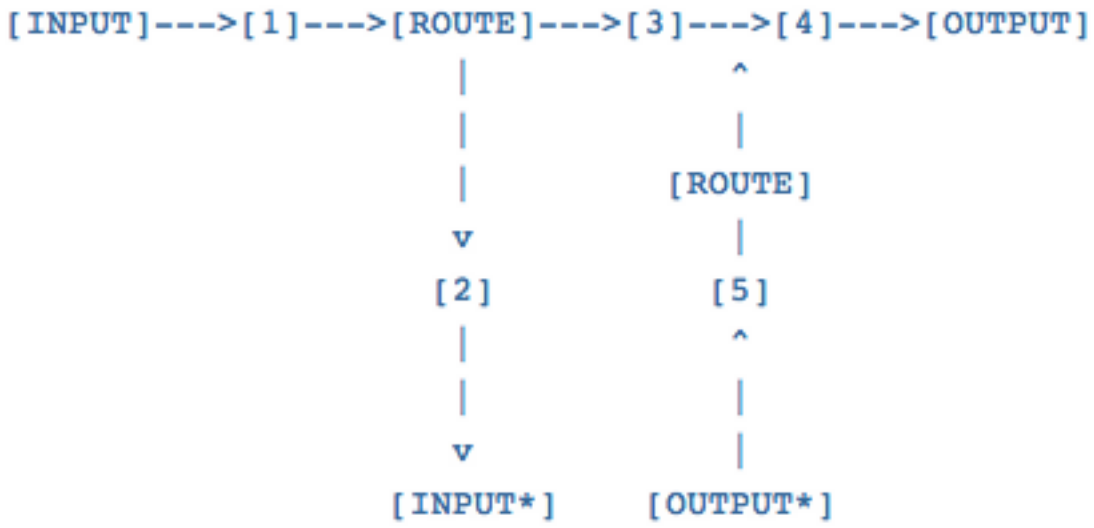
1. Packet filtering and capture: netfilter hooks and how they work
2. Packet re-injection: the NF_HOOK macro
3. Hrtimers

1. Packet Filtering and Capture: netfilter hooks and how they work

Thankfully the ability to hook into the kernel to observe and manipulate network traffic is already provided by netfilter hooks. Unfortunately, nearly all documentation and examples were written in the early to mid 2000s and the implementation has change subtly (or in some cases extremely) since then.

Netfilter hook locations

The following figure illustrates where these hooks exist in the life cycle of an incoming and outgoing packet:



[1] NF_INET_PRE_ROUTING,
[2] NF_INET_LOCAL_IN,
[3] NF_INET_FORWARD,
[4] NF_INET_LOCAL_OUT,
[5] NF_INET_POST_ROUTING

NOTE: Hooks are defined in /linux/netfilter.h

The hook that I used was NF_INET_PRE_ROUTING to capture the packets because this hook occurs before any in-depth packet processing occurs. Two things should be noted about this hook. The first is that checksums are validated before this point. The second is that I'm fairly certain that this is the same hook that wireshark uses to capture data. This became a slight issue during testing when wireshark would see the packets as they came in.

The hook function

The function that will be executed when the kernel hook is reached must have the following signature:

unsigned int packet_filter_hook(unsigned int hooknum, struct sk_buff *skb, const struct net_device *in,
const struct net_device *out, int (*okfn)(struct sk_buff*))

Hooknum: this value maps to one of the five previously discussed. The reason for this is that the registered function can handle multiple hooks at a time.
Skb: is the standard socket buffer that contains the packet data.

In and out: these are pointers to the device that is responsible for direction of the dataflow. Only one of these will ever have a value at a time. For example an incoming packet will have a value for in, but out will be NULL. An outgoing packet will have a value for out, but not for in.

Okfn: Initially the some of the message board posts I read regarding the okfn function pointer said not to be concerned about it and that it was not really used. This could not be further from the truth. The okfn function pointer points to the next function that should be executed once the hook is done executing and the packet is accepted or queued.

The return value for this function must be one of the following:

1. NF_ACCEPT: Accept the packet
2. NF_DROP: Drop it. Tells the kernel that no processing is being done within the hook on the packet
3. NF_STOLEN: Tells the kernel that we are going to do processing on the packet and to continue doing it's own processing
4. NF_QUEUE: Queue up the packet to be processed
5. NF_REPEAT: Re-inject the packet just before this hook is called so that it can be processed again

Obviously the return value that is best in this case is NF_STOLEN.

Registering the hook function

To register the hook function a nf_hook_ops object must be created and then registered. For example:

```
netfilter_ops.hook          =     packet_filter_hook;
netfilter_ops.pf            =     PF_INET;
netfilter_ops.hooknum       =     NF_INET_PRE_ROUTING;
netfilter_ops.priority      =     NF_IP_PRI_FIRST;
nf_register_hook(&netfilter_ops);
```

Hook: hook is the name of the hook function that is to be called when the hook is reached
Pf: is the protocol family this hook is going to be monitoring. In this case the IPv4 family of protocols.
Hooknum: this value maps to one of the five previously discussed
Priority: Execution priority for this hook. This is for the cases when we are managing multiple hooks within one module

2. Packet re-injection: the NF_HOOK macro

The NF_HOOK macro is a elegant way of having netfilter hooks process packets. However, it can be used to re-inject packets into the normal packet processing. The signature for NF_HOOK is:

```
#define NF_HOOK(pf, hook, skb, indev, outdev, okfn) (okfn)(skb)
```

Pf: the protocol family which is already known because it is the only protocol family that this module is dealing with (at this time)

Hook: Which hook we wish to insert the packet at. This actually ended up being a bit tricky since if the packet was re-inserted back at NF_INET_PRE_ROUTING the packet would be processed infinitely. Therefore, NF_INET_LOCAL_IN was chosen as re-injection point. Initially one might think that this will cause problems if the packet should be routed elsewhere and wasn't meant for the current computer, however okfn takes care of this.

Skb: this again is the socket buffer that contains all of the packet information and then some. This struct is fairly complicated, however a macro skb_copy() is provided to save us the nightmare of having to preform a deep copy ourselves

Indev/outdev: can be retrieved from pf

Okfn: okfn is the function that will be executed if the packet is accepted. It should be noted that the NF_HOOK macro is what the kernel uses to check for hooks, especially within net/ipv4/ip_input.c which has no externned function. This is why we need to know what okfn. By saving that function pointer, we can at a later time re-inject the packet where it would have gone if we had allowed it to process as usual, regardless of which hook we re-inject it at.

In light of the NF_HOOK macro, skb_copy, and okfn it is safe to say that the only data members that need to be saved so that the packet can be delayed and then re-injected are skb and okfn.

3. Hrtimers

High resolution timers offers up finer granularity than normal timers. The implementation is very similar however, hrtimers do not use jiffies, they use nanoseconds. There were two main hurdles I had to overcome to work with hrtimers.

1. Passing data to the callback function
2. Canceling and cleaning up the hrtimers and related structures since memory space in the kernel is very limited

Passing data to the callback function

The standard timer structure has data members for both the callback function and data to be passed to the callback function. While hrtimers store the callback function, there is no place to pass data to that function. In the case of this project that data is skb and okfn. What I ended up doing was creating the following struct:

```
struct packet_data{
	struct hrtimer delay_timer;
	struct sk_buff *skb;
	int  (*okfn)(struct sk_buff *);
};
```

By embedding the hrtimer into the struct, container_of can be called on the hrtimer to retrieve the packet_data. Thereby in a round about way allowing the needed data to be passed to the callback function, send_delayed_packet.

Canceling and cleaning up

Normally this wouldn't be a problem however, these hooks are event driven and if you cancel a timer from its callback function a segmentation fault will occur. In addition, as long as the hrtimer exists within the timers red-black tree the struct it is embedded in shouldn't be freed. As such, I use the follow two variables to point to the current timer and packet_data. When the next hrtimer times out or the module is unloaded the pointed to hrtimer and packet_data will be cleared or freed.

```
static struct hrtimer * hrtimer_to_cancel;
static struct packet_data * pd_to_free;
```

4. Conclusions

This was an incredibly interesting project and I'm really glad that I got to do it. It is also one of those projects where number of lines of code (less than 160 including comments and spacing) does not reflect the amount of work that went into it.