# COMP689 Project Report

## A Distributed ICMP Network Responder (DINR) Service

Craig Tomkow
6-10539 85 Ave
Edmonton, AB T6E 2K5
Canada
ctomkow@gmail.com

## ABSTRACT

Global name servers can be misused as an external network connectivity failover test using the Internet control message protocol (ICMP). The aim of the project is to create the underlying infrastructure for a distributed ICMP network responder service for network connectivity testing. The project focuses on two main components, linux ICMP reply performance and distribution of the service behind a single network address. The final design requires a kernel bypass to achieve sufficient ICMP performance while the single address distribution employs the network routing concept of anycast. Linux ICMP performance testing demonstrates that a virtual environment for the ICMP service is insufficient. Furthermore, simple kernel modifications on real hardware was determined to be inadequate for the service. Ultimately, the ICMP performance testing led to the conclusion that a kernel bypass was required. While kernel bypass is part of the overall design, it could not be implemented due to network hardware constraints. Finally, anycast was implemented within a virtual network environment demonstrating the concept of a distributed service using a single address. The project establishes a framework design through implementation and testing for the distributed ICMP network responder service.

## Keywords

Distributed Systems, ICMP, Linux Kernel, Anycast, Networking

## 1 Introduction

Internet infrastructure is typically designed for a specific goal which enables global scalability. A core Internet infrastructure was identified as being used for an unintended purpose. The domain name system (DNS) infrastructure's primary function is to provide a distributed database of name and address tuples, resolving names based on queries. DNS provides an effective name resolution service that has scaled globally to serve the whole Internet infrastructure. However, with the success of DNS came the unintended abuse of the service for other purposes. Due to the prevalence and reliability of DNS, organizations and network administrators have begun using reachability to external DNS infrastructure as a test of outside network connectivity. This is done using an Internet control message protocol (ICMP) echo request towards a public DNS name server, such as Google's 8.8.8.8 name servers, and expecting an ICMP echo reply from the operating system kernel. This can be performed through the classic network troubleshooting program, `ping`. Beyond simple network connectivity testing, network administrators and products [10] have used the ICMP echo test for wide area network (WAN) failover. This has resulted in many organizations networks relying on a WAN failover mechanism that misuses another Internet service.

The aim of the project is to create the underlying infrastructure for a distributed ICMP network responding service (DINR); the infrastructure must be able to scale globally while presenting only one network address for administrators to point their WAN failover to. Deploying a proof of concept distributed service is out of scope for this project. The project contains two main components; the ICMP echo response service and the distribution of servers behind a single address. Due to hardware dependencies for enabling reliable ICMP echo response, ICMP performance was explored as far as possible until hardware limitations poised unsolvable issues within the scope and time-frame of this project. However, future work will explore the next steps to fully develop the infrastructure. Finally, the ICMP echo response service was attempted first in a virtual server environment, then attempted with real hardware for performance reasons. Based on the chosen method for distributing the service behind a single address, the development of the second project component was done in a virtual network environment for convenience.

## 2 Background Reading

To assist in the design of the project, various research was required. The first project component addresses ICMP echo response performance. The original project concept was to develop a cloud deployable server for rapid scalability and management. To determine cloud viability, an understanding of how ICMP echo requests are handled in the linux operating system was required. Of all the ICMP message types, echo requests processed and responded to by the kernel [15]. Received messages are not passed to a user space process. To improve a linux system's ICMP echo response, changes need to occur within the kernel. While, some kernel modifications in cloud environments are possible for higher performance networking, such as Amazon Web Services enhanced networking [2], it is not common among all cloud providers which limits flexibility. Furthermore, the hardware is still virtualized which limits performance. The second issue with a cloud deployment is the ease of distributing the service behind a single address. While cloud environments have software defined load balancing services that can easily distribute traffic to servers across the globe, such as Google's cloud load balancer [5], further research revealed

that the load balancing service only operated at layer 4 and above (of the open systems interconnection (OSI) network model). The implication is that the load balancer only distributes user datagram protocol (UDP) and transmission control protocol (TCP) traffic. This poses a problem as ICMP operates at layer three; not dependant on UDP or TCP. Ultimately, it was clear that the DINR service could not be cloud deployable.

## 2.1 Linux Kernel Networking

Once it was determined that the service could not be deployed in a cloud infrastructure, further investigation was done on the two core aspects of the service. As determined earlier, ICMP echo response is a purely linux kernel process. To explore linux kernel modifications, an understanding of the packet flow within linux was required. *Linux Kernel Networking* by Rami Rosen [14] was a critical resource in understanding linux networking. The basic process is as follows: an Ethernet packet is received by the network interface card (NIC) and placed in a receive (Rx) queue. The Rx queue is first in first out (FIFO), similar to many common buffer implementations. The NIC software driver receives the packet and performs layer two processing. In older network drivers and kernel, each packet arrival caused a central processing unit (CPU) interrupt, informing the CPU to process the packet. This caused significant CPU overhead and degradation of performance during high network activity. This was changed with the introduction of the new API (NAPI) in the 2.6 linux kernel. The network driver is now able to buffer multiple packets and the kernel polls the network device driver periodically reducing the number of CPU interrupts required. Once the kernel polls the network driver for packets, it copies each packet into a socket buffer; the **`sk_buff`** struct. The buffer is used to store each packet, allowing for easy reference while it is processed at each layer of the OSI stack. After the packet has progressed from layers two through four, it is then handed to the user space process that is bound to a specific socket. This message passing between the kernel and user space is handled by the **`netlink`** protocol.

## 2.2 Address Transparency

To create a globally distributed service behind a single address, options other than a cloud infrastructure were explored. There are two broad concepts for distributing the service, application level distribution and network layer distribution. A common method of globally load balancing a service is using DNS. A straightforward, albeit naive method is to have multiple address (A) records mapped to the same name. These A records would each represent a single DINR server. Each A record would have the same domain name but different internet protocol (IP) addresses. In this way, when a request to resolve the domain name is received by the name server, it will provide a different IP address for each request, load balancing the requests in a round-robin order. While this works to distribute load across multiple servers that are geographically dispersed, it has flaws. DNS does not know whether a server is available or not, therefore it may direct traffic to a non-functioning server. Furthermore, it does not know where a server may be located geographically which could result in directing traffic to the other side of the world. To effectively distribute a service globally and transparently behind a single address, it can be best served using layer three network routing. The basic concept is to rely on the routing of the Internet to send the traffic to the nearest DINR server available. The concept is called anycast and it is widely recognized as a core mechanism for globally distributed systems.

# 3 Design

## 3.1 Service Specifications

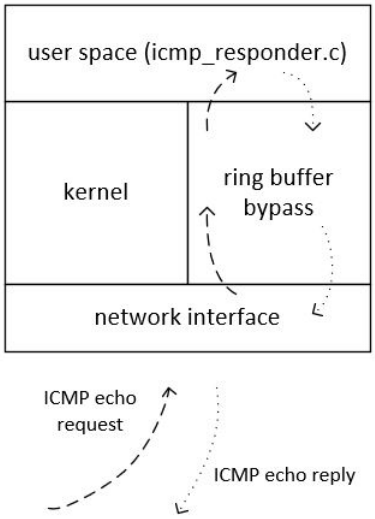
To properly inform the project design and implementation, core service requirements require detailing. The linux kernel's ICMP echo response performance target is to quantifiably improve upon the default linux kernel performance. No specific target is specified; the main reason for this is due to the difficulty in estimating a theoretical network load in a globally distributed system. Traffic demands are highly dependant on current usage, server numbers, location, and Internet routing. Having said this, a guiding direction was determined. A linux system should successfully process enough traffic, $n$ number of packets per second (pps), to saturate a standard one gigabit per second (Gbps) NIC. Basing a requirement on a 1 Gbps NIC allows for the deployment of standard and relatively inexpensive server hardware. While this is a seemingly modest aim, high pps (e.g. one million pps (Mpps)) can be difficult to achieve. Also, note that certain techniques required for achieving a high pps may scale beyond 1 Gbps.

The second core project requirement is to hide the DINR service behind a single network address. The perceived popularity of Google's 8.8.8.8 address for testing network connectivity is in part due to its single, memorable IP address. To achieve this project requirement, empirical testing is required to prove reachability to different DINR servers targeting one address from different client locations. While the two main goals of the project are ICMP response performance and distributed reachability through a single address, there are other specifications applicable to a distributed system such as extensibility, security, and distribution of system administration. These requirements are relegated to future work.

## 3.2 System Design

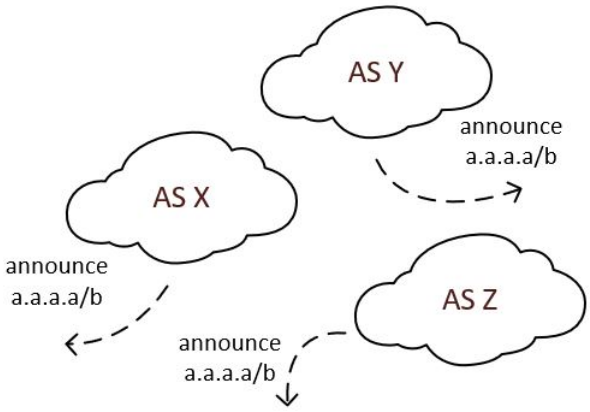
### *3.2.1 Kernel Performance*

The distributed system design is split into two main components. The design for increasing ICMP echo response performance evolved over the course of the project. Initially, the understanding was that sufficient software modifications of the default linux kernel would result in a saturated 1 Gbps network interface. However, as the project implementation and testing proceeded, it was clear that achieving that level of performance was difficult. Further research discovered a concept called kernel bypass [9]. The concept involves either a separate kernel module or re-compiling the kernel to pass traffic from the network driver directly to user space for processing. The technique effectively side-steps the kernel entirely. During the process of research, testing, and kernel modification, it was clear that kernel bypass is required. It was determined that if the kernel was not bypassed, software changes to the kernel would be needlessly complex. Ultimately, the design below focuses on a kernel bypass design.




### 3.2.2 *Single Address Design*

Distributed address transparency is a more straightforward design than the ICMP response specifics. Address transparency employs the concept of anycast. Anycast is the routing of messages to the nearest resource. This concept fits perfectly for the DINR service; a client's ICMP echo request should be routed to the nearest resource, reducing latency and potential for lost traffic. Global anycast is based on the idea of advertising the same IP prefix (e.g. 50.0.0.0/8) from multiple geographically diverse areas. Internet routing, using the border gateway protocol (BGP), is relied on to forward the traffic to the nearest source. The simplicity of the concept allows for scalability. Furthermore, due to anycast requiring only advertising one prefix via BGP to an upstream service provider, it is easily deployed by multiple organizations (autonomous systems (AS)). This enables distributing the administrative overhead to multiple organizations. The advertising of the same IP prefix by multiple organizations only typically requires a letter of authority (LOA) by the owner of the IP prefix to permit a third party to advertise the prefix.




# 4 Testing and Implementation

## 4.1 Kernel Network Performance

To be able to effectively test whether there was an increase in packet per second processing of ICMP echo request packets, there are a number of details that needed resolving. An appropriate packet sending tool was required to test the limits of receiving a high volume of ICMP packets. Also, monitoring of some sort was required to determine if there was packet loss. Additionally, calculations were required to determine what the theoretical maximum packets per second is on a 1 Gbps link. Finally, a layout of the various test environments are given with the related testing results.

### 4.1.1 *Performance Testing Tools*

#### 4.1.1.1 Transmitting Test Traffic

As part of the implementation, test ICMP echo request traffic needed to be generated at a sufficient speed that the receiving server would start dropping traffic due to congestion. The naive approach was to use the `ping` program, native to most operating systems. The `-f` flag was used on a linux machine to flood pings (ICMP echo requests) as fast as possible toward the receiving server. However, there was concern regarding the speed that ping could actually generate and send the packets. Other programs were explored that specializes in crafting and sending packets at speed. `netperf` and `iPerf` were researched, however, those programs did not support crafting ICMP packets.
Having searched and not found an appropriate program, a program was created to send ICMP echo requests. The program was written in the C programming language, purely for speed. Creating the program allowed for flexibility as well. It provided a way for timestamping, sending a specific number of packets, and specifying the preferred length of a packet. The core of **`icmp_send.c`** works as follows. First, a raw socket is created with the type **`IPPROTO_ICMP`**.

```
sockfd = socket(AF_INET, SOCK_RAW,
                IPPROTO_ICMP);
```

Then, the destination address is specified for the **`sockaddr_in`** struct.

```
dest_addr.sin_family = AF_INET;
inet_pton(AF_INET, "192.168.1.150",
          &dest_addr.sin_addr);
```

The ICMP header type is set to 8 with a code of 0 for an ICMP echo request.

```
echo_req = (struct icmphdr *)
           (icmp_packet);
echo_req->type = 8;
echo_req->code = 0;
```

Finally, the echo request packets are transmitted within a simple loop.

```
for (seq_num = 0; seq_num < 100000000;
     seq_num++) {
   sendto(sockfd, icmp_packet,
          icmp_packet_len, 0,
          (struct sockaddr *) &dest_addr,
          sizeof(dest_addr));
```

While these details are the core of sending an ICMP packet; there is other supporting code not shown such as: checksum, timestamp, packet size, and memory allocations. The full code can be seen on the following github page or in appendix C.

```
ctomkow/dinr/blob/master/icmp_send.c
```

The **icmp_send.c** program succeeds in crafting and sending ICMP echo requests as fast as possible. While it is admittedly only single-threaded, it is trivial to invoke the program multiple times for parallelism.

#### 4.1.1.2 Monitoring Traffic

Once sending echo requests at speed was achieved, a way to detect whether packets were being lost or not was needed. Again, a naive approach was used initially by monitoring traffic in realtime to detect dropped packets. `tcpdump` was first used to monitor all incoming and outgoing traffic, logging a difference in received/sent traffic. It was soon discovered that `tcpdump` was too slow and could not buffer and process all the traffic quickly enough. This resulted in `tcpdump` being a bottleneck itself. Further research was done into other utilities that could poll or monitor the network stack. Programs such as `dropwatch` and `perf` were explored, however they suffered from the same problem. For example, `perf` only polls the network stack 1000/sec [6], too slow for testing that deals with thousands of packets per second. It was obvious that inline measuring via `tcpdump`, or polling of the kernel's network stack was insufficient. This led to the idea of indirect measuring of the sender and receiver while directly measuring at line-rate in-between the two servers.

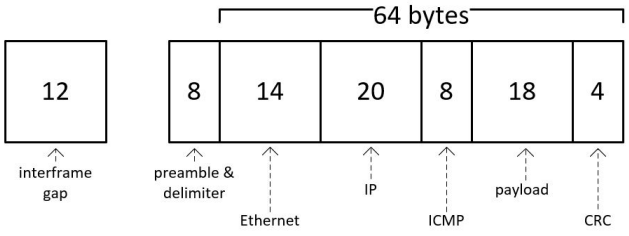
The concept of indirect measurement is simple. The sender, `srv1`, sends $n$ packets and verifies they were sent by examining network counters. The receiver, `rcv`, verifies the number of packets received, $m$, by viewing the network counters. The second step is to verify sent packets by `rcv`, in response to `srv1`. This was accomplished again by looking at the sent and received network counters for `rcv` and `srv1`, respectively. In measuring this way, packet loss was determined by the delta between the sent and received counters. Furthermore, gathering the statistics from both `srv1` and `rcv` allowed for determining where the packet loss occurred, either at the sender or receiver. There are a couple linux utilities available to monitor counters, `ethtool` and `netstat`. `netstat` was primarily used in conjunction with `watch`, to monitor the change in counters as the testing was taking place.

To verify the monitoring process, a quick explanation of how the counters are incremented is necessary. A packet is received on the NIC and passed to the kernel's **ip_rcv()** procedure; the IP counters are then incremented. When the packet processing is finished at the IP layer, it is passed to the **ip_local_deliver_finish()** procedure. The procedure notes the ICMP header and invokes the **icmp_rcv()** procedure. Initially, the ICMP counters are incremented and then the **icmp_echo()** procedure is invoked. **icmp_echo()** changes the ICMP header to an echo reply (type 0, code 0) and invokes the **icmp_reply()** procedure, which transmits the reply packet and increments the network counters. It's important to note that a packet could be dropped anywhere along the path before hitting the kernel's ICMP processing code and it won't be clear where or why it was dropped; only that it was.

A final note regarding network monitoring. Inline monitoring was achieved as well with no dropped packets. A non-blocking line-rate Cisco switch was used as the network fabric between the servers. This allowed for monitoring the network bandwidth and packet counters while not causing undesirable traffic loss. While it only captured performance numbers at a specific point in time during testing, it provides additional data to correlate with the network counter numbers.

### 4.1.2 *Packet per Second Calculations*

To provide context for the testing metrics, packet per second calculations are required. The smallest packet possible was used for calculations - this created a worst case scenario for the linux kernel. When it comes to host networking performance, pps metrics is more interesting than raw bandwidth as pps causes significantly more strain on the kernel, network interface, and CPU. With that said, the smallest possible Ethernet frame according to the Institute of Electrical and Electronics Engineers (IEEE) 802.3 specification is 64 bytes. Ethernet is the last encapsulation header before modulation via the physical layer transceiver (PHY), therefore, the size of the echo request packet including all protocol headers will be 64 bytes. To calculate the maximum number of packets possible on a 1 Gbps interface, additional bytes must be considered at the Ethernet layer; 7 bytes for the preamble and 1 byte for the frame delimiter. Furthermore, Ethernet specifies a minimum interframe gap, the sending of an idle pattern by the PHY, which is required between each sent frame [20]. The standard interframe gap time is 96 bit times (12 bytes).




Determining this was critical in creating **icmp_send.c**. The exact size of the packet has to been known when creating the packet. The following line of code specifies, in bytes, the size of the ICMP header and payload.

```
icmp_packet_len = 26;
```

By having the ICMP packet's total size be 26 bytes (8 byte ICMP header and 18 byte payload), 38 bytes were left for the IP header (20 bytes) and Ethernet frame header/trailer (18

bytes). In total, this creates a 64 byte packet. To calculate the maximum number of pps for a network interface, the formula is:

$$\frac{\frac{net\_if\_bps}{8}}{pkt\_size + preamble + delimiter + gap}$$

Therefore, the calculation for a 1 Gbps network interface is straightforward.

$$\frac{\frac{1{,}000{,}000{,}000}{8}}{64 + 7 + 1 + 12} = 1,488,095.24$$

The theoretical maximum that `srv1` can send and `rcv` can respond is ∼1.488 Mpps.

### *4.1.3 Testing*

#### 4.1.3.1 Virtual Environment

The initial testing was performed in a virtual environment on a standard desktop out of convenience. The sender and receiver Ubuntu servers were created within VMware workstation 14 Pro. The desktop had a quad port gigabit network interface card which allowed for binding each virtual server to a physical gigabit interface. A home router/switch was used to interconnect the virtual machines (VM's).
There were initial concerns regarding a virtual environment earlier when exploring a cloud environment, however, utilizing this setup to start helped establish a testing process. The host desktop has a 2.5Ghz quad-core CPU and 16GB of random access memory (RAM). Each virtual machine was allocated one CPU core, 2GB of RAM, and one physical gigabit NIC. To begin with, a 1,000,000 echo requests were sent from `srv1`.

*srv1* invokes **icmp_send.c**

```
2018:03:24 16:42:00.061
2018:03:24 16:42:22.349
numbr sent: 1000000
```

The receiver is shown below responding to all echo requests without dropping traffic.

*rcv* output of *netstat -sw*

```
IcmpMsg:
        InType3: 80
        InType8: 1000000
        OutType0: 1000000
        OutType3: 80
```

It is clear that the sender could not generate echo requests fast enough to overwhelm the receiving linux kernel. This was due to the virtual environment. A simple calculation of the packets sent is performed.

$$\frac{pkt\_sent}{time\_delta} = 44,867pps$$

The virtual environment, hosted on a standard desktop PC, is insufficient for testing. Further attempts were made to increase the sending rate. Invoking the process twice on `srv1` only succeeded in causing the runtime to double while keeping the pps the same. The hardware requirements for the sender had to be increased. The sender VM was allocated two CPU's and invoked two processes, each pinned to a CPU. As seen in the images below, the sender machine could not keep up with sending the 2,000,000 echo requests in a virtual environment. However, even though the sender only sent 1,313,947 packets, the receiver only responded to 462,864 packets. The receiver was overwhelmed, albeit with inconsistent sender behaviour. It was clear that real hardware was required to perform these tests.

*srv1* output of *netstat -sw*

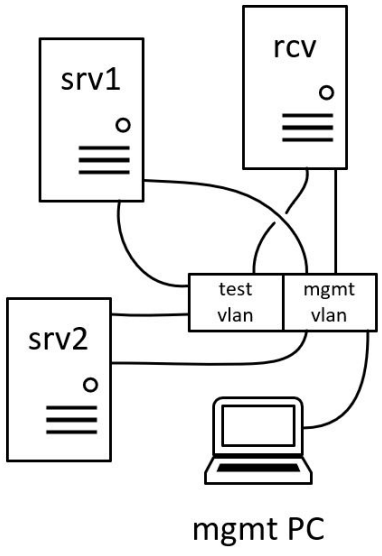
```
IcmpMsg:
        InType0: 462864
        InType3: 80
        OutType3: 80
        OutType8: 1313947
```

*rcv* output of *netstat -sw*

```
IcmpMsg:
        InType3: 80
        InType8: 462864
        OutType0: 462864
        OutType3: 80
```

#### 4.1.3.2 Physical Environment

Due to the virtual environment exhibiting sub-optimal performance (unsurprisingly), an environment with bare-metal server hardware was set up. The topology was setup to mirror the virtual environment, however, a managed Cisco switch interconnected the servers. Furthermore, an out-of-bound network was set up on a secondary network interface of the servers to allow for logging into the servers. The management network was segregated on a separate virtual local area network (VLAN).




For this testing, the sender stayed consistent by sending 1,000,000 packets.

*srv1* output of *netstat -sw*

```
IcmpMsg:
        InType0: 999965
        InType3: 404
        OutType3: 404
        OutType8: 1000000
```

Testing on real hardware significantly increased the sending rate to 192,344 pps. Even with a small test of one million packets, only 999,965 ICMP echo replies (type 0) were received demonstrating the beginning of packet loss at the receiver. Therefore, at around 192,000 pps, the linux kernel can not respond quickly enough. Finally, to verify that the packet loss was indeed occurring on the receiver end, the network statistics below show the `rcv` kernel only sending 999,965 echo replies while indeed receiving all 1,000,000 packets.

*rcv* output of *netstat -sw*

```
IcmpMsg:
        InType3: 724
        InType8: 1000000
        OutType0: 999965
        OutType3: 724
```

Further baseline testing was performed, however, the measurement was taken from the inline Cisco switch. For this test, the number of packets sent was increased to one billion, allowing for ongoing traffic observation. Looking at the statistics from the switch interface connected to `rcv`, a comparison can be made to earlier tests. Below is an image of the switch port statistics, showing an output rate (from `srv1` to `rcv`) of 388,727 pps. The input rate is 268,332 pps (from `rcv` to `srv1`). Beyond the packet loss at the receiver, it highlights a performance increase in sending/receiving sustained traffic. Finally, note that the switch port traffic monitoring reports only the bits transmitted that comprise a packet, implying exclusion of the interframe gap.

switch port stats for *rcv*

```
input rate 137387000 bits/sec, 268332 packets/sec
output rate 199029000 bits/sec, 388727 packets/sec
```

### 4.1.4 Performance Investigation

The testing demonstrated packet loss with bare-metal hardware. While the sender could not saturate a 1 Gbps interface with 64 byte packets on a single core process (only ∼200 Mbps), the receiver was unable to respond quickly enough regardless. In an attempt to resolve this issue, further investigation was required. The processor usage of the receiver was the first place to investigate. The linux utility `top` was used to observe the receiver's processor utilization during sustained network traffic.

*rcv* output of *top*

```
top - 19:47:02 up  1:08,  3 users,  load average: 0.99, 0.57, 0.24
Tasks: 124 total,   2 running, 122 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  0.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,100.0 si,  0.0 st
%Cpu2  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  4040872 total,  3504532 free,    71128 used,   465212 buff/cache
KiB Swap:   998396 total,   998396 free,        0 used.  3725512 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
   13 root      20   0       0      0      0 R  99.7  0.0   3:53.62 ksoftirqd/1
 2865 ctomkow   20   0   41728   3840   3256 R   0.3  0.1   0:00.36 top
    1 root      20   0   37664   5736   4036 S   0.0  0.1   0:03.54 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:00.00 ksoftirqd/0
```

Parsing top results immediately uncovers an issue. **`ksoftirqd/1`** process is utilizing all of CPU1 and no other CPUs were being used. To further investigate, **`ksoftirqd`** research was required. In essence, **`ksoftirqd`** is a per-CPU kernel thread software interrupt, triggered by the NICs message signaled interrupt (MSI), that polls via NAPI for network packets to process. Due to the large amount of packets per second being received, the network stack is interrupting CPU1 constantly for packet processing to the point that the processor can not keep up. Consequently, the network driver drops packets due to insufficient processor time. Note that the software interrupt is triggered by the NICs interrupt (MSI) when packets are received, therefore, if there is inadequate processor time to process the interrupts, packets will be dropped at the network driver before it even reaches core kernel code. This is why the counters do not directly count the dropped packets.

The next step was to investigate how to spread the interrupt load across multiple CPUs. Further research revealed the concept of receive side scaling (RSS) [17] which involves increasing the number of Rx queues in a NIC. By having multiple Rx queues in the NIC, combined with improved interrupts (MSI-X), each Rx queue interrupt can be directed to a specific CPU. This is primarily a hardware-based solution and due to working with older hardware, the NICs installed on the servers only had a single Rx queue. While the existing hardware constrained implementation, a software solution was desired. The concept of receive packet steering (RPS) [12] was discovered; a software implementation of RSS. The basic concept of RPS is as follows. When a packet is initially copied to the network stack via **`netif_receive_skb()`** procedure, a hash is calculated over the packet (in this case, a 2-tuple of the source and destination address). The hash result is used to determine which CPU should process the packet. To configure RPS, the linux kernel needed configuring. For Ubuntu linux server 16.04.4 LTS with kernel 4.4, the configuration file required a bit mask specifying which CPUs can process packets from the specified queue. For all four CPUs to be eligible for use, the bit mask of 00001111 (f in hexadecimal) is used.

```
echo f >
/sys/class/net/eno1/queues/Rx-0/rps_cpus
```

To properly test the kernel tweak, two senders were used, `srv1` and `srv2`, to ensure the hashing was used. We can see by the image below that the number of packets per second being sent by `rcv` is 361,927. An increase from the earlier testing.

switch port stats for *rcv*

```
input rate 185306000 bits/sec, 361927 packets/sec
output rate 390214000 bits/sec, 762136 packets/sec
```

Furthermore, `top` shows other processors being utilized, however minimal.

*rcv* output of *top*

```
top - 09:40:30 up 5 min,  1 user,  load average: 0.96, 0.53, 0.23
Tasks: 122 total,   2 running, 120 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,100.0 si,  0.0 st
%Cpu1  :  0.0 us,  0.0 sy,  0.0 ni, 98.9 id,  0.0 wa,  0.0 hi,  1.1 si,  0.0 st
%Cpu2  :  0.3 us,  0.0 sy,  0.0 ni, 99.3 id,  0.3 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  4040872 total,  3822028 free,    64700 used,   154144 buff/cache
KiB Swap:   998396 total,   998396 free,        0 used.  3766144 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
    3 root      20   0       0      0      0 R 100.0  0.0   1:34.82 ksoftirqd/0
    6 root      20   0       0      0      0 S   0.3  0.0   0:00.04 kworker/u16:0
    1 root      20   0   37628   5660   4000 S   0.0  0.1   0:03.05 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kthreadd
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
```

To verify that software interrupts are being spread across CPUs, the following command was used.

```
watch -d cat /proc/softirqs
```

Note the NET_RX software interrupts on both CPU0 and CPU1. Only two servers are sending echo requests, therefore, the hashing is spreading the interrupts between two CPUs only. Theoretically, the performance should increase as traffic is received from unique sources.

*rcv* output of *watch -d cat /proc/softirqs*

```
                    CPU0       CPU1       CPU2       CPU3
          HI:          3          2          1          0
       TIMER:      51756      78897      20330       7557
      NET_TX:         41         62        728          8
      NET_RX:    2026228    2345506     228809      23473
```

Finally, a comparison should be done to the stock kernel with two senders sending sustained echo requests, to verify the results observed. Using a stock kernel, the inline measurement indicates 268,665 pps while the software interrupts were only served by CPU3. Receive packet steering on a single Rx queue NIC with only two sources of traffic improved performance by ∼35%.

switch port stats for *rcv*

```
input rate 137557000 bits/sec, 268665 packets/sec
output rate 402748000 bits/sec, 786615 packets/sec
```

*rcv* output of *watch -d cat /proc/softirqs*

```
                    CPU0       CPU1       CPU2       CPU3
          HI:          0          0          0          6
       TIMER:       7067       4190       5740      50904
      NET_TX:         80         87         13         15
      NET_RX:      17980      17904      17967     832414
```

### *4.1.5 Kernel Bypass*

After exploring network kernel performance, it is clear that in addition to performance being hardware dependant, significant kernel modifications may be required. Due to the complexity of the kernel code, further research explored the concept of kernel bypass. The concept is straightforward - instead of having the traffic depend on kernel processing, software hooks directly into the network card via a modified network driver forwarding traffic directly to user space. A number of kernel bypass implementations exist: PF_RING [11], data plane development kit (DPDK) [19], and netmap [13]. Each implementation has a similar end goal while employing slightly different techniques. PF_RING for example, can be installed as a kernel module in Ubuntu, greatly simplifying setup without the need for kernel compiling. Kernel bypass appears to be the way to proceed, allowing for ICMP echo requests to be directly passed to user space and responded to. However, as a kernel bypass implementation was explored, it became clear that it is highly hardware dependant. Due to the implementation requiring modified network drivers, only modern, higher end NICs were supported in all implementations. Therefore, implementing kernel bypass was not possible for the project.

While kernel bypass is not possible at this time, it is still worth discussing the possible implementation of an **`icmp_responder.c`** program. The basic flow of traffic would be as follows. An ICMP echo request is received on the NIC which is copied to the network driver, the modified driver would then pass the packet directly to the kernel bypass software. The bypass copies all packets into a ring buffer. From here, the packet is accessible via an exposed application programming interface (API) for copying the packet to a user space **`icmp_responder.c`** program. Within the program, it would read the ICMP header and verify that the packet is a type 8 ICMP echo request, change the ICMP header to type 0 echo reply, and send the packet out via the API for transmission. The program would be multi-threaded (or multi-process) to achieve CPU parallelism, effectively solving the problem. The kernel bypass implementation is relegated to future work.

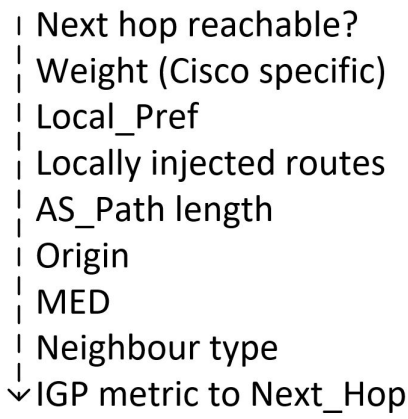
### *4.1.6 Summary*

Testing and investigation of linux kernel performance was performed. When dealing with low level kernel networking, simple software changes are not enough. Proper hardware is important to achieve a certain level of performance, whether through multi-queue NICs or network driver support for kernel bypass. While saturation of a 1 Gbps NIC was not achieved in this project, a significant foundation of knowledge was gained through the learning process. The area of linux kernel network performance is a large area of study, worthy of future work.

## 4.2 Address Transparency

To distribute the DINR service behind a single IP address, anycast was implemented in a virtual network environment to demonstrate the concept. Anycast is the routing of traffic to the nearest resource. In essence, the same prefix is advertised from multiple areas while relying on a routing protocol to inform routers to forward traffic toward the nearest resource. In terms of a global anycast service, a specific prefix is advertised to an upstream Internet service provider (ISP) via the border gateway protocol (BGP) from multiple geographic locations. The idea being that due to BGP being a path vector protocol, the shortest number of autonomous system (AS) hops will be installed into a routers routing table. This ensures that the router is forwarding traffic toward the closest advertised prefix. With BGP's path selection algorithm, as long as the details that come before AS_Path

length are the same, BGP will use the fewest number of autonomous system hops to determine forwarding. In general, the fewer the AS's between you and the destination, the closer it is.




### *4.2.1 Anycast Implementation*

To clarify the concept, anycast was implemented in the graphical network simulator-3 (GNS3) program using virtualized Cisco 3750 routers. The topology (seen in appendix A) simulates service providers, enterprise networks, and the DINR networks. As the topology shows, there are two DINR locations (DINR_A and DINR_B) advertising the same prefix of 50.0.0.0/8 via BGP to their upstream Internet service providers ISP_A and ISP_B, respectively. Each ISP will advertise the prefix to their neighbours, the upstream ISP (ISP_Z) and the downstream enterprise. If we examine each of the enterprise's BGP routes using the command `show ip bgp`, we should only see the AS path towards the DINR service that is closest to them.

Enterprise_A BGP table

| | Network | Next Hop | Metric | LocPrf | Weight | Path |
|---|---|---|---|---|---|---|
| *> | 1.0.0.0 | 172.16.0.17 | | | 0 | 4 2 1 i |
| *> | 2.0.0.0 | 172.16.0.17 | | | 0 | 4 2 i |
| *> | 3.0.0.0 | 172.16.0.17 | | | 0 | 4 2 1 3 i |
| *> | 4.0.0.0 | 172.16.0.17 | 0 | | 0 | 4 i |
| *> | 5.0.0.0 | 172.16.0.17 | | | 0 | 4 2 1 3 5 i |
| *> | 11.0.0.0 | 172.16.0.17 | | | 0 | 4 11 i |
| *> | 20.0.0.0 | 0.0.0.0 | 0 | | 32768 | i |
| *> | 21.0.0.0 | 172.16.0.17 | | | 0 | 4 2 1 3 5 21 i |
| *> | 50.0.0.0 | 172.16.0.17 | | | 0 | 4 12 i |

Enterprise_B BGP table

| | Network | Next Hop | Metric | LocPrf | Weight | Path |
|---|---|---|---|---|---|---|
| *> | 1.0.0.0 | 172.16.0.21 | | | 0 | 5 3 1 i |
| *> | 2.0.0.0 | 172.16.0.21 | | | 0 | 5 3 1 2 i |
| *> | 3.0.0.0 | 172.16.0.21 | | | 0 | 5 3 i |
| *> | 4.0.0.0 | 172.16.0.21 | | | 0 | 5 3 1 2 4 i |
| *> | 5.0.0.0 | 172.16.0.21 | 0 | | 0 | 5 i |
| *> | 11.0.0.0 | 172.16.0.21 | | | 0 | 5 3 1 2 4 11 i |
| *> | 20.0.0.0 | 172.16.0.21 | | | 0 | 5 3 1 2 4 20 i |
| *> | 21.0.0.0 | 0.0.0.0 | 0 | | 32768 | i |
| *> | 50.0.0.0 | 172.16.0.21 | | | 0 | 5 13 i |

By examining the BGP tables for each enterprise for prefix 50.0.0.0/8, it is clear that the AS path points towards the nearest DINR service for each enterprise. For example, enterprise_A BGP table shows the first hop being AS 4, then the last hop being AS 12. This clearly matches the topology map, showing ISP_A as AS 4, and DINR_A as AS 12. This is anycast proper.

To verify that anycast is indeed working correctly, traceroute is used to prove the routers are forwarding traffic correctly. For example, ENT_A-testPC will initiate a traceroute to the DINR destination address of 50.50.50.50. Then, ENT_B-testPC will traceroute to the same destination address, 50.50.50.50. The traceroutes below show traffic being forwarded to the nearest DINR server.

Enterprise_A PC traceroute to 50.50.50.50

```
trace to 50.50.50.50, 8 hops max (ICMP), press Ctrl
 1   20.0.0.1   10.587 ms  9.579 ms  10.198 ms
 2   172.16.0.17   30.304 ms  31.000 ms  30.481 ms
 3   172.16.0.30   41.265 ms  39.965 ms  29.907 ms
 4   50.50.50.50   61.283 ms  50.270 ms  40.735 ms
```

Enterprise_B PC traceroute to 50.50.50.50

```
trace to 50.50.50.50, 8 hops max (ICMP), press Ctrl
 1   21.0.0.1   5.023 ms  10.653 ms  9.132 ms
 2   172.16.0.21   19.891 ms  19.241 ms  19.757 ms
 3   172.16.0.34   29.496 ms  39.360 ms  40.225 ms
 4   50.50.50.50   50.733 ms  51.663 ms  51.256 ms
```

To contrast this behaviour, a generic centralized service was setup at address 11.0.0.10. Performing a traceroute from Enterprise_B's PC results in traffic traversing the entire topology, an undesirable result when discussing global distribution.

Enterprise_B traceroute to 11.0.0.10

```
trace to 11.0.0.10, 8 hops max (ICMP), press Ctrl+C
 1   21.0.0.1   7.418 ms  9.643 ms  9.382 ms
 2   172.16.0.21   20.034 ms  20.176 ms  19.664 ms
 3   172.16.0.9   30.750 ms  31.088 ms  29.771 ms
 4   172.16.0.5   40.591 ms  41.443 ms  41.137 ms
 5   172.16.0.2   50.648 ms  51.022 ms  50.825 ms
 6   172.16.0.14   60.619 ms  61.517 ms  61.452 ms
 7   172.16.0.26   73.159 ms  71.694 ms  80.946 ms
 8   11.0.0.10   92.686 ms  104.239 ms  103.789 ms
```

Finally, a curious mind might be wondering how ISP_Z views the 50.0.0.0/8 prefix due to multiple advertisements of the same address space. As seen in the BGP route table below, it shows the router learning both advertisements. However, note the '>' for only one of the routes for 50.0.0.0/8, it has been chosen as the 'best' route according to BGP's path selection algorithm.

ISP_Z BGP table

| | Network | Next Hop | Metric | LocPrf | Weight | Path |
|---|---|---|---|---|---|---|
| *> | 1.0.0.0 | 0.0.0.0 | 0 | | 32768 | i |
| *> | 2.0.0.0 | 172.16.0.2 | 0 | | 0 | 2 i |
| *> | 3.0.0.0 | 172.16.0.6 | 0 | | 0 | 3 i |
| *> | 4.0.0.0 | 172.16.0.2 | | | 0 | 2 4 i |
| *> | 5.0.0.0 | 172.16.0.6 | | | 0 | 3 5 i |
| *> | 11.0.0.0 | 172.16.0.2 | | | 0 | 2 4 11 i |
| *> | 20.0.0.0 | 172.16.0.2 | | | 0 | 2 4 20 i |
| *> | 21.0.0.0 | 172.16.0.6 | | | 0 | 3 5 21 i |
| * | 50.0.0.0 | 172.16.0.2 | | | 0 | 2 4 12 i |
| *> | | 172.16.0.6 | | | 0 | 3 5 13 i |

## 5 Future Work

The distributed ICMP network responder service was developed in two parts; ICMP kernel performance and address transparency. While progress was made regarding both areas, further work is desired. ICMP network performance is lacking. While minimal gains were achieved, proper hardware is required to proceed with testing default kernel performance with receive side steering, while future development can be done on an **`icmp_responder.c`** program utilizing kernel bypass. Furthermore, **`icmp_responder.c`** could be expanded to account for security issues, implementing basic filtering to prevent malicious use of the service. Finally, while anycast was fully understood and implemented in a virtual network environment, actual implementation of anycast in the real world is preferred. The foundations of the

DINR service were explored throughout the project, with potential for significant future work.

## 6 Conclusion

The aim of the distributed ICMP network responder service is to expose a single IP address for receiving ICMP echo requests while reliably responding with ICMP echo replies. Due to the potential scale of the service, only parts of the service could be explored in this project. Anycast routing was successfully deployed within a virtual network environment, achieving the desired routing of traffic to a single IP address. While the anycast implementation was successful, further work can explore deploying anycast in the real world. ICMP performance within the linux kernel was investigated. The ability for a server to respond to a high quantity of traffic, specifically a high packet per second, is critical for the service to succeed. While quite a bit of network testing was performed, only minimal performance gains were achieved falling short of the ∼1.488 Mpps target. However, a solid foundation of knowledge for further research was established. The project was successful in identifying key components required for a high performance network stack such as multi-queue network interface cards for multi-CPU processing of network traffic. Additionally, to implement a kernel bypass to increase network performance, supported network hardware is required for the kernel bypass network drivers to function. Finally, additional work on an ICMP responder process was identified as a core component of the service to integrate with the kernel bypass as well as address security concerns. Overall, the DINR service has a solid foundation based on the work performed in the project.

## 7 References


[1] Abley, J., et al. Operation of Anycast Services. RFC 4786, IETF Secretariat, Dec. 2006.

[2] Amazon. Enhanced networking on linux, 2018.

[3] Brouer, J. Network stack challenges at increasing speeds. In *Linux.conf.au* (2015), LCA2015.

[4] Carnegie Mellon University. in_cksum, 2018.

[5] Google LLC. Google cloud load balancing - high performance, scalable load balancing on google cloud platform, 2018.

[6] kernel.org. Linux kernel profiling with perf, 2015.

[7] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language*. Prentice Hall, New Jersey, 2008.

[8] Majkowski, M. How to receive a million packets per second. Tech. rep., Cloudflare, June 2015.

[9] Majkowski, M. Kernel Bypass. Tech. rep., Cloudflare, September 2015.

[10] Meraki. Connection monitoring for wan failover. Tech. Rep. 1320, Cisco Systems, Inc., 2018.

[11] ntop. PF_RING - high-speed packet capture, filtering and analysis, 2018.

[12] Red Hat. Receive packet steering (rps), 2018.

[13] Rizzo, L. netmap - the fast packet i/o framework, 2018.

[14] Rosen, R. *Linux Kernel Networking - Implementation and Theory*. Apress, New York, 2014.

[15] Stevens, W. R. *TCP/IP Illustrated, Volume 1 The Protocols*. Addison-Wesley, Massachusetts, 1994.

[16] Stevens, W. R., et al. *Unix Network Programming, Volume 1: The Sockets Networking API*. Addison-Wesley, Massachusetts, 2003.

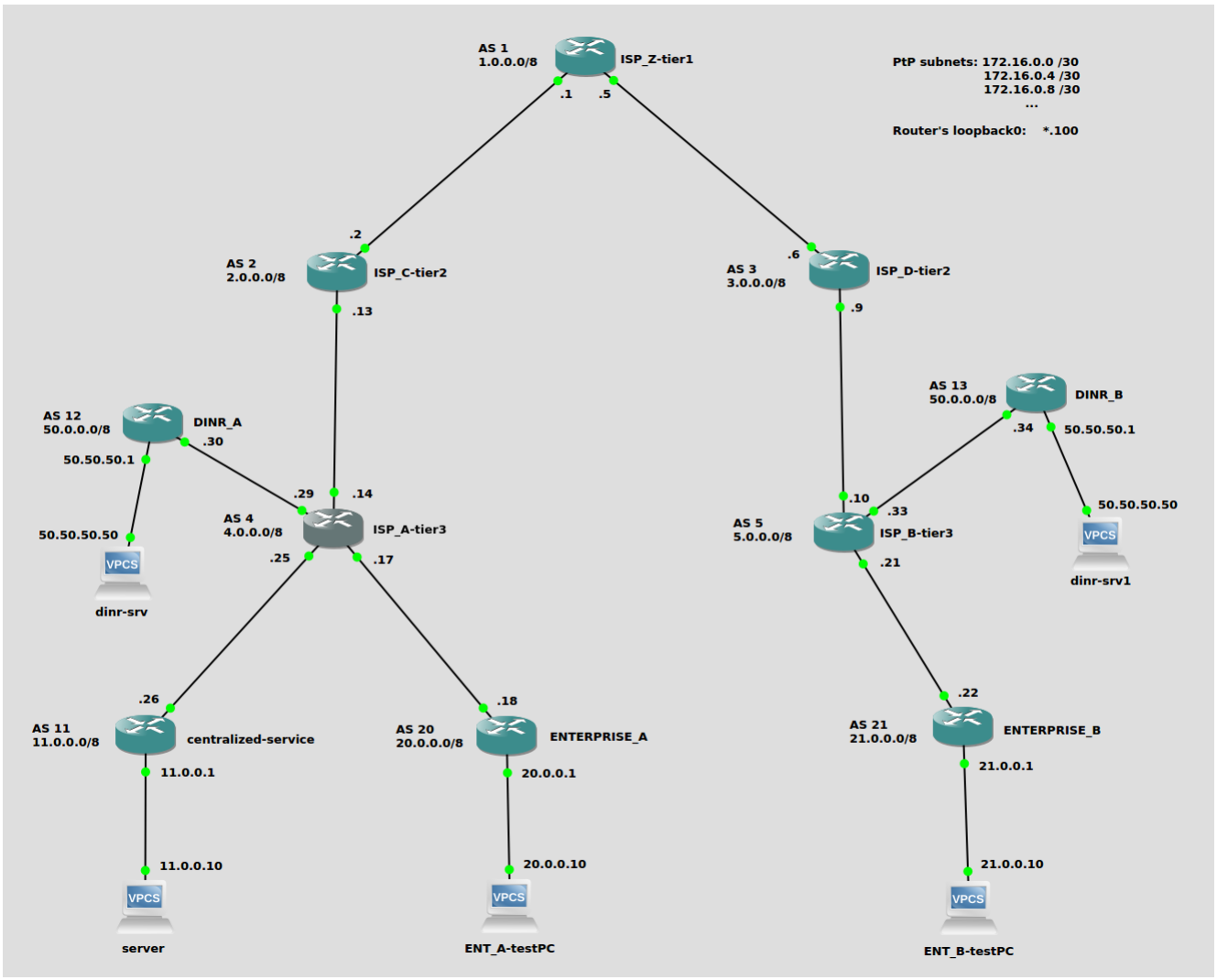
[17] T. Herbert and W. Bruijn. Scaling in the linux networking stack, 2018.

[18] Tanenbaum, A., and Steen, M. V. *Distributed Systems Principles and Paradigms*. Pearson India Education Services, Noida, 2013.

[19] The Linux Foundation Projects. DPDK - data plane development kit, 2018.

[20] Wikipedia. Interpacket gap, 2017.

# APPENDIX
# A
## Anycast GNS3 Topology



## B

### Router BGP Configurations

```
!! DINR_A
!
router bgp 12
 bgp log-neighbor-changes
 neighbor 172.16.0.29 remote-as 4
 !
 address-family ipv4
  neighbor 172.16.0.29 activate
  auto-summary
  no synchronization
  network 50.0.0.0
 exit-address-family
!
```

```
!! DINR_B
!
router bgp 13
 bgp log-neighbor-changes
 neighbor 172.16.0.33 remote-as 5
 !
 address-family ipv4
  neighbor 172.16.0.33 activate
  auto-summary
  no synchronization
  network 50.0.0.0
 exit-address-family
!
```

```
!! ISP_A
!
router bgp 4
 bgp log-neighbor-changes
 neighbor 172.16.0.13 remote-as 2
 neighbor 172.16.0.18 remote-as 20
 neighbor 172.16.0.26 remote-as 11
 neighbor 172.16.0.30 remote-as 12
 !
 address-family ipv4
  neighbor 172.16.0.13 activate
  neighbor 172.16.0.18 activate
  neighbor 172.16.0.26 activate
  neighbor 172.16.0.30 activate
  auto-summary
  no synchronization
  network 4.0.0.0
 exit-address-family
!
```

```
!! ISP_B
!
router bgp 5
 bgp log-neighbor-changes
 neighbor 172.16.0.9 remote-as 3
 neighbor 172.16.0.22 remote-as 21
 neighbor 172.16.0.34 remote-as 13
 !
 address-family ipv4
  neighbor 172.16.0.9 activate
  neighbor 172.16.0.22 activate
  neighbor 172.16.0.34 activate
  auto-summary
  no synchronization
  network 5.0.0.0
 exit-address-family
!
```

```
!! ENTERPRISE_A
!
router bgp 20
 bgp log-neighbor-changes
 neighbor 172.16.0.17 remote-as 4
 !
 address-family ipv4
  neighbor 172.16.0.17 activate
  auto-summary
  no synchronization
  network 20.0.0.0
 exit-address-family
!
```

```
!! ENTERPRISE_B
!
router bgp 21
 bgp log-neighbor-changes
 neighbor 172.16.0.21 remote-as 5
 !
 address-family ipv4
  neighbor 172.16.0.21 activate
  auto-summary
  no synchronization
  network 21.0.0.0
 exit-address-family
!
```

## C

### icmp_send.c

```c
/*
Craig Tomkow
March 25, 2018
COMP689 - Advanced Distributed Systems
A Distributed ICMP Network Responder (DINR) service
icmp_send.c
-This program sends ICMP echo requests (type 8, code 0) as fast as possible.
 It is used to test the limits of a linux kernel responding to small (64 byte) requests.
*/


#include <stdio.h> // printf()
#include <stdlib.h> // malloc()
#include <arpa/inet.h> // inet_pton (specific to Ubuntu) - also exposes <netinet/in.h>
#include <netinet/ip_icmp.h> // icmphdr struct
#include <unistd.h> // sleep()
#include <time.h> // localtime(), strftime()
#include <sys/time.h> // gettimeofday(), timeval struct
#include <string.h> // memset()

// declare functions
void print_time();
unsigned short in_cksum(unsigned short *, int);

int main() {

    int        sockfd; // raw_sock file descriptor
    char      *icmp_packet; // pointer to memory address for packet data
    int        icmp_packet_len; // integer defining icmp packet size in bytes
    long int   seq_num; // sequence number of sent ICMP echo request
    struct     icmphdr *echo_req; // pointer to icmphdr in memory
    struct     sockaddr_in dest_addr; // struct used for send to des addr

    sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP); // create raw socket

    // fill in sockaddr_in struct with destination address details
    dest_addr.sin_family = AF_INET;
    inet_pton(AF_INET, "192.168.1.150", &dest_addr.sin_addr);

    // specify payload and packet size in bytes, reserve memory
    icmp_packet_len     = 26; // ICMP_HDR + PAYLOAD (8 + 18)
    icmp_packet         = malloc(icmp_packet_len);

    // 0's for packet buff (checksum is calculated only for icmp header)
    memset (icmp_packet, 0, icmp_packet_len);

    // reserve and specify ICMP header details
    echo_req = malloc(sizeof(struct icmphdr)); // reserve memory for icmphdr
    echo_req = (struct icmphdr *) (icmp_packet) ; // icmp_packet cast as icmphdr
    echo_req->type     = 8; // icmp echo request, type 8 code 0
    echo_req->code     = 0;
    echo_req->checksum = 0;

    echo_req->checksum = in_cksum((unsigned short *)echo_req, sizeof(struct icmphdr));

    // print start time
    print_time();

    // send ICMP echo requests
    for (seq_num = 0; seq_num < 100000000; seq_num++) {
        sendto(sockfd, icmp_packet, icmp_packet_len, 0, (struct sockaddr *)
        &dest_addr, sizeof(dest_addr));
    }

    // print end time
    print_time();

    printf("numbr sent: %ld\n", seq_num);

}
```

```c
void print_time() {

    // Time vars
    char      time_buff[26]; // time buffer
    int       ms; // store milliseconds
    struct    tm *time_info; // a date and time struct (only sec precision)
    struct    timeval time; // sec and microsec time struct

    gettimeofday(&time, NULL); // get time

    ms = time.tv_usec / 1000.0; // convert from microseconds to milliseconds
    if (ms >= 1000) {
        ms -= 1000;
        time.tv_sec++;  // if millisecond is >= 1000, incr second
    }

    time_info = localtime(&time.tv_sec); // assigns time to time_info struct
    strftime(time_buff, 26, "%Y:%m:%d %H:%M:%S", time_info); // formats time

    printf("%s.%03d\n", time_buff, ms); // prints time

}

// cksum src: cs.cmu.edu/afs/cs/academic/class/15213-f00/unpv12e/libfree/in_cksum.c
unsigned short in_cksum(unsigned short *addr, int len) {

    int             nleft     = len;
    int             sum       = 0;
    unsigned short *w         = addr;
    unsigned short  answer    = 0;

    while (nleft > 1) {
        sum   += *w++;
        nleft -= 2;
    }

    if (nleft == 1) {
        *(unsigned char *)(&answer) = *(unsigned char *)w;
        sum += answer;
    }

    sum    = (sum >> 16) + (sum & 0xffff);
    sum   += (sum >> 16);
    answer = ~sum;
    return(answer);

}
```