TCP/IP Troubleshooting for Systems Administrators

TCP/IP Troubleshooting for Systems Administrators

PUBLISHED BY:
Darren Hoch
hochdarren@gmail.com

For a collection of all papers by Darren Hoch:

http://www.ufsdump.org

Table of Contents

1.0 Introduction to TCP/IP Troubleshooting

The core of modern IT infrastructure is based on multiple interconnected networks. Without networks, almost all of the infrastructure that powers the way we conduct our very lives would be useless. Networks rely on the use of common protocols. A networking protocol is a set of communications standards that define how two systems pass information.

The TCP/IP protocol has evolved into the most common networking protocol on the earth. Everything from the home network built on the $50 Taiwanese built router to the trunks of major telecom providers use this protocol. All 21st century computing tasks such as social networking, SOA architecture, and compute clusters rely on TCP/IP to transmit the data required to accomplish tasks.

As with anything built by a human, that thing will eventually break down and require troubleshooting, maintenance, and repair. The TCP/IP protocol is no exception. The focus of this paper is to use the most common open computing platform, Linux, to develop a disciplined approach to troubleshooting the TCP/IP network. The Linux platform provides a robust set of network debugging tools that provide insight into the most common network problems.

This paper breaks troubleshooting into two sections: **reactive** and **proactive**.

1.1 Reactive Troubleshooting

Reactive troubleshooting identifies problems in network communication that result in errors. Examples of these errors include: hosts or networks unreachable and applications unavailable. The reactive section breaks troubleshooting into 4 categories based on protocol: **Ethernet/ARP**, **IP/ICMP**, **TCP/UDP**, and **application**. For each category, the paper will describe the important components of the protocol, how to use tools to monitor for errors, how to correlate the output from the tools, and the corrective actions that need to be taken.

Example tools include `tcpdump`, `tethereal`, `dsniff`, `Wireshark`, `netstat`, `arp`, `route` and `ifconfig`. Example case studies include troubleshooting ARP, IP routing, TCP states, and common application protocols.

1.2 Proactive Troubleshooting

Proactive troubleshooting identifies problems that result in poor performance. Examples include slow response times or limited throughput. The proactive section examines the network as a whole, identifying all of the parts of the network that contribute to poor performance.

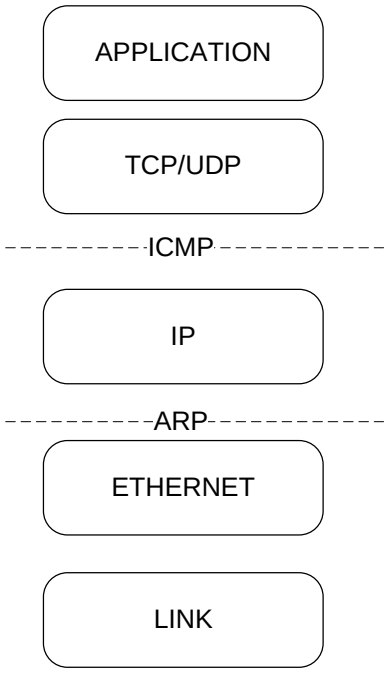
Example tools include `ethtool`, tcptrace, `iptraf`, `netperf`, and `ntop`. Example case studies include saturated network links and improper network settings.

This paper assumes that the reader has a base understanding of how to use a Linux system, basic TCP/IP knowledge, and experience using the basic network monitoring tools.

2.0 Review of the TCP/IP Stack

The TCP/IP stack consists of 5 layered protocols. Of these 5 layers, a system administrator troubleshoots the top 4. The link layer consists of cables, wires, and the actual signals bits that are passed along the wire. A TCP/IP stack running on a network device organizes these bits into 4 transportation protocols. Each transportation protocol has a specific responsibility in the flow of data.

Figure 1: Standard "5 Layer" Model



The protocol responsibilities are listed below:

- **ETHERNET** – defines the address of the next physical host that processes the data, either another host on the network or a router to another network
- **IP** – defines the address of the next network host that should process the data, either a host on the current network or a router to another network
- **TCP/UDP** – defines the application port that of the receiving host on the network
- **APPLICATION** – contains the data payload that must be processed by the application

3.0 Troubleshooting with Packet Capture

Packet capture is the process of collecting and analyzing TCP/IP packets on an Internet connection. Network cards have the ability to listen to both traffic destined only to their specific device and traffic destined to all devices. The mode of capturing all traffic (also know as "sniffing" or "snooping") on the wire is known as **promiscuous mode**. The ability to for a system administrator to listen to traffic in this mode depends on two things: **privileges** and **location on the network**.

Almost all packet sniffing tools require super user privileges the commands. A non-routing node in promiscuous mode monitors traffic to and from other nodes within the same collision domain (for Ethernet and Wireless LAN) or ring (for Token Ring or FDDI) and broadcast domain.

A collision domain includes all the nodes that share a single wire. Deprecated equipment , such as network hubs, place all nodes on a single wire. Modern switches build "virtual wires" between two end points, eliminating collisions on a network and the ability to snoop in promiscuous mode.

Network switches are used to combat malicious use of promiscuous mode and therefore limit a node's collision domain to broadcast traffic and unicast traffic from the host. To capture traffic, a systems administrator needs access to a router that will monitor all traffic that it routes. If the router is not accessible, a node may have access to a physical port on a switch that displays a copy of all traffic (also known as a port mirror).

3.1 `libpcap` Based Tools

The `libpcap` library is a system independent interface for user level packet capture. This library provides a high-level API for packet capture, which creates a simple packet capture abstraction layer for developing tools. As a result, many Linux based packet capture utilities use the `libpcap` interface as their underlying packet capture engine. Due to the portability of this code, all utilities that use the `libpcap` library share the same syntax. The most common utilities that use the `libpcap` library are `tcpdump` and `ethereal`.

3.1.0 The `tcpdump` Utility

The `tcpdump` utility is the most common packet capture utility for Linux based systems. It ships as a standard package with every common Linux distribution and is often included in the default installation or available via standard package managers. The source code for the utility is located at `http://www.tcpdump.org`.

The `tcpdump` command in and of itself produces volumes of data on a busy network.


```
box:~# tcpdump
tcpdump: verbose output suppressed, use v or vv for full protocol decode
listening on eth0, linktypeEN10MB (Ethernet), capture size 96 bytes
10:19:23.042904 IP 192.168.1.60.ssh > 192.168.1.102.hpidsadmin: P
51744:51888(144) ack 561 win 15232
10:19:23.043898 IP 192.168.1.60.ssh > 192.168.1.102.hpidsadmin: P
51888:52032(144) ack 561 win 15232
10:19:23.046280 IP 192.168.1.60.ssh > 192.168.1.102.hpidsadmin: P
52032:52176(144) ack 561 win 15232
10:19:23.048005 IP 192.168.1.102.hpidsadmin > 192.168.1.60.ssh: . ack
52304 win 16960
10:19:23.049376 IP 192.168.1.60.ssh > 192.168.1.102.hpidsadmin: P
52304:52448(144) ack 561 win 15232
10:19:23.051003 IP 192.168.1.102.hpidsadmin > 192.168.1.60.ssh: . ack
52576 win 16688
10:19:23.052322 IP 192.168.1.60.ssh > 192.168.1.102.hpidsadmin: P
52576:52720(144) ack 561 win 15232
10:19:23.054006 IP 192.168.1.60.ssh > 192.168.1.102.hpidsadmin: P
52720:52864(144) ack 561 win 15232
10:19:23.056292 IP 192.168.1.60.ssh > 192.168.1.102.hpidsadmin: P
52864:53008(144) ack 561 win 15232
```

By default, the `tcpdump` command attempts to resolve the IP address of all connections inbound to the node. If the node does not have access to a DNS server, the packets captured by `tcpdump` are delayed until resolution. Another issue is that `tcpdump` defaults to the primary interface on a node. If a systems administrator desires to monitor any other interface, he or she must specify it explicitly.

In the following example, the "`-i`" specifies another Ethernet interface besides the default (`eth0`). Also included is the "`-n`" option which turns of host and port resolution.

```
box:~# tcpdump -ni eth1
tcpdump: verbose output suppressed, use v or vv for full protocol decode
listening on eth1, linktype EN10MB (Ethernet), capture size 96 bytes

19:47:22.935554 IP 192.168.1.105.32783 > 67.110.253.165.25: P
3432387228:3432387265(37) ack 2742259796 win 63712 <nop,nop,timestamp
239675 1064682926>

19:47:22.967508 IP 67.110.253.165.25 > 192.168.1.105.32783: P 1:54(53)
ack 37 win 1984 <nop,nop,timestamp 1064879093 239675>
```

Line spaces are inserted into the previous output for the sake of clarity and explanation.

Each line in the previous output describes a single packet. The line items are described below.

- **Real Time:** `19:47:22.967508`
- **Source IP Address:** `IP 67.110.253.165.25`
- **Direction of Packet Flow:** `>`
- **Destination Address:** `192.168.1.105.32783:`

- **TCP Flags:** P
- **TCP Source SYN Number:** 1:
- **Next TCP SYN Number**: 54(53) # original SYN (1) + payload (53) = next SYN (54)
- **TCP ACK NUMBER:** ack 37
- **TCP Window Size:** win 1984
- **TCP Options:** <nop,nop,timestamp 1064879093 239675>

In some cases, traffic needs to be monitored now and reviewed offline or the volume of traffic to be monitored is too great for a standard terminal window buffer. The option "-w" writes a packet to a file and the option "-r" reads the capture file.

```
box:~# tcpdump –w /tmp/tcp.out –ni eth1
tcpdump: listening on eth1, linktype
EN10MB (Ethernet), capture size 96 bytes
46 packets captured
46 packets received by filter
0 packets dropped by kernel

box:~# tcpdump –r /tmp/tcp.out –ni eth1
reading from file /tmp/tcp.out, linktype EN10MB (Ethernet)

19:56:07.190888 IP 192.168.1.105.32783 > 67.110.253.165.993: P
3432387731:3432387768(37) ack 2742260475 win 63712 <nop,nop,timestamp
292100 1065283060>

19:56:07.227315 IP 67.110.253.165.993 > 192.168.1.105.32783: P 1:54(53)
ack 37 win 1984 <nop,nop,timestamp 1065403449 292100>

<<snip>>
```

3.1.1 The tethereal Utility

Just like tcpdump, ethereal is based on the libpcap interface. There are two main versions of ethereal. There is the text version called "tethereal" and the GUI based version called "Wireshark". The text based version is very similar in syntax to the tcpdump command syntax. Once again, this is because they use the same underlying libpcap engine.

```
box:~# tethereal –w /tmp/ethereal.out –ni eth1
Capturing on eth1
0.327450 192.168.1.105 > 67.110.253.165 TLS Application Data
0.361175 67.110.253.165 > 192.168.1.105 TLS Application Data
0.361220 192.168.1.105 > 67.110.253.165 TCP 32783 > 993 [ACK]
Seq=37 Ack=53 Win=63712 Len=0 TSV=389797 TSER=1066380554
0.363460 192.168.1.105 > 67.110.253.165 TLS Application Data
0.410951 67.110.253.165 > 192.168.1.105 TLS Application Data

box:~# tethereal –r /tmp/ethereal.out
6 2.543822 192.168.1.105 > 67.110.253.165 TLS Application Data
7 2.593330 67.110.253.165 > 192.168.1.105 TLS Application Data
```



```
8 2.593375 192.168.1.105 > 67.110.253.165 TCP 32783 > imaps [ACK]
Seq=37 Ack=53 Win=63712 Len=0 TSV=412045 TSER=1066603077
9 2.595989 192.168.1.105 > 67.110.253.165 TLS Application Data
```

The default output of `tethereal` is less detailed than the `tcpdump` output. The differences are listed below:

- **Packet numbering** – the first column of output shows the packet number relative to the order of the capture
- **Relative time** – the time (in seconds) the packet was captured relative to the start of the capture (0.0 seconds)
- **Application summary data** – all packets summarized by application type (`TLS Application Data` , for example)

The `tethereal` binary is a symbolic link to `tshark`, the "official" command line name.

3.1.1 Using the `dsniff` Utility

The `dsniff` command is an older suite of utilities written by Dug Song (`http://monkey.org/~dugsong/dsniff/`). Unlike the previously mentioned utilities, `dsniff` takes packet capture one level further. Using the underlying `libpcap` engine, `dsniff` takes the packets captured and attempts to report something a little more useful. The `dsniff` program is one of many utilities in the `dsniff` package. The standard `dsniff` command will attempt to capture and replay all unencrypted sessions including: FTP, telnet, SMTP, IMAP, and POP.

The following example demonstrates how to use `dsniff` to an ftp sessions:

```
# dsniff -ni eth0
dsniff: listening on eth0
-----------------
07/13/08 14:35:37 tcp 192.168.1.102.3832 -> 192.168.1.60.21 (ftp)
USER darren
PASS darren
```

The `dsniff` output provides the protocol, IP address, port, and credentials of the FTP session.

The latest official release on the author's website is 2.3. The newest release maintained by the community is 2.4 and is available in many of the "extras" repositories of popular Linux distributions.

3.2 Using Filter Expressions

It may be easy to identify specific traffic streams on small or idle networks. It will be much harder to accomplish this on large WAN or saturated networks. The ability to use filter expressions is extremely important in these cases to cut

out unwanted "noise" packets from the traffic in question. Fortunately, both the `libpcap` based utilities and the snoop utility all use the same filter syntax. There are many ways to filter traffic in all utilities, the most common filters are by `port`, `protocol`, and `host`. The following example tracks only SMTP traffic and host `192.168.1.105`:

```
[root@targus ~]# tcpdump –ni eth0 port 25 and host 192.168.1.105
tcpdump: verbose output suppressed, use v or vv for full protocol decode
listening on eth0, linktype EN10MB (Ethernet), capture size 96 bytes

21:13:26.905262 IP 192.168.1.105.32899 > 192.168.1.220.smtp: S
1903904803:1903904803(0) win 5840 <mss 1460,sackOK,timestamp 722613
0,nop,wscale 0>

[root@targus ~]# tethereal –ni eth0 port 25 and host 192.168.1.105
Capturing on eth0
0.000000 192.168.1.105 > 192.168.1.220 TCP 32900 > 25 [SYN] Seq=0 Ack=0
Win=5840 Len=0 MSS=1460 TSV=729689 TSER=0 WS=0
```

The previous two examples leverage the most common types of filter expressions:

- **`host`** – capture specific host traffic
- **`port`** – capture specific port traffic
- **`net`** – capture all traffic on a specific network
- **`and/or/not`** – standard operators to combine expressions
- **`src/dst`** – specify source and destination hosts

The following example tracks all outbound traffic to the 192.168.1.0 network from a single source host of :

```
# tcpdump -ni eth0 src host 192.168.1.102 and dst net 192.168.1.0/24
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
15:29:14.154742 IP 192.168.1.102.4071 > 192.168.1.60.ssh: . ack 1877201002 win
16192
15:29:14.282674 IP 192.168.1.102.4071 > 192.168.1.60.ssh: . ack 129 win 16064
15:29:14.383722 IP 192.168.1.102.4071 > 192.168.1.60.ssh: . ack 257 win 17520
15:29:14.484762 IP 192.168.1.102.4071 > 192.168.1.60.ssh: . ack 385 win 17392
15:29:14.583309 IP 192.168.1.102.4071 > 192.168.1.60.ssh: . ack 513 win 17264
15:29:14.681853 IP 192.168.1.102.4071 > 192.168.1.60.ssh: . ack 641 win 17136
15:29:14.782434 IP 192.168.1.102.4071 > 192.168.1.60.ssh: . ack 769 win 17008
```

There are other cases where an administrator may want to capture all but certain types of traffic. Specifically, a lot of "noise" can be made if one is trying to run a packet capture while logged into the remote host. Much of the traffic generated will be the control traffic back to that host. The following example shows how to filter the `ssh` control traffic to and from the control connection (`192.168.1.105` connected to `192.168.1.220` as `root`) and all DNS traffic.

```
[root@targus ~]# who
root pts/2 Jun 1 21:30 (192.168.1.105)

[root@targus ~]# tcpdump –ni eth0 not host 192.168.1.105 and not port 53
tcpdump: verbose output suppressed, use v or vv for full protocol decode
listening on eth0, linktype EN10MB (Ethernet), capture size 96 bytes
```



```
21:32:07.692524 IP 216.93.214.50.51606 > 192.168.1.220.ssh: S
2550704294:2550704294(0) win 5840 <mss 1460,sackOK,timestamp 431608120
0,nop,wscale 2>

21:32:07.692596 IP 192.168.1.220.ssh > 216.93.214.50.51606: S
729994889:729994889(0) ack 2550704295 win 5792 <mss 1460,sackOK,timestamp
111008380 431608120,nop,wscale 2>

21:32:07.796911 IP 216.93.214.50.51606 > 192.168.1.220.ssh: . ack 1 win
1460 <nop,nop,timestamp 431608221 111008380>
```

The above example demonstrates how a system administrator logs into a remote system (192.168.1.220**), monitors the connections to and from that system, and excludes the SSH control traffic coming from the administrator's workstation (**192.168.1.105**).**

4.0 Troubleshooting Ethernet

The Ethernet layer enables a system to send packets to other systems on the same network. Ethernet addresses are 48 bit addresses that are hard coded on individual NIC cards. A system with multiple NIC cards has multiple Ethernet addresses.

Ethernet addresses are presented in octets and can be viewed using the –e switch in `tcpdump`. The following example shows the Ethernet address of two nodes communicating within a network (192.168.1.102 and 192.168.1.60):

```
# tcpdump -e -q -ni eth0 port 22
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
16:49:53.193805 00:0c:29:a3:84:72 > 00:13:ce:d8:23:0c, IPv4, length 166:
192.168.1.60.ssh > 192.168.1.102.4121: tcp 112

16:49:53.194633 00:0c:29:a3:84:72 > 00:13:ce:d8:23:0c, IPv4, length 166:
192.168.1.60.ssh > 192.168.1.102.4121: tcp 112

16:49:53.195783 00:13:ce:d8:23:0c > 00:0c:29:a3:84:72, IPv4, length 60:
192.168.1.102.4121 > 192.168.1.60.ssh: tcp 0
```

Alternately, the `tethereal` utility provides the ability to print the entire Ethernet frame in detail. The following example displays the Ethernet portion of a single packet:

```
# tethereal -nVi eth0 -c 1
Running as user "root" and group "root". This could be dangerous.
Capturing on eth0

Frame 1 (134 bytes on wire, 134 bytes captured)
    Arrival Time: Jul 13, 2008 16:56:03.702211000
    [Time delta from previous captured frame: 0.000000000 seconds]
    [Time delta from previous displayed frame: 0.000000000 seconds]
    [Time since reference or first frame: 0.000000000 seconds]
    Frame Number: 1
    Frame Length: 134 bytes
    Capture Length: 134 bytes
    [Frame is marked: False]
    [Protocols in frame: eth:ip:tcp:ssh]
Ethernet II, Src: 00:13:ce:d8:23:0c (00:13:ce:d8:23:0c), Dst: 00:0c:29:a3:84:72
(00:0c:29:a3:84:72)
    Destination: 00:0c:29:a3:84:72 (00:0c:29:a3:84:72)
        Address: 00:0c:29:a3:84:72 (00:0c:29:a3:84:72)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
        .... ..0. .... .... .... .... = LG bit: Globally unique address (factory
default)
    Source: 00:13:ce:d8:23:0c (00:13:ce:d8:23:0c)
        Address: 00:13:ce:d8:23:0c (00:13:ce:d8:23:0c)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
        .... ..0. .... .... .... .... = LG bit: Globally unique address (factory
default)
    Type: IP (0x0800)
```


Most Ethernet traffic is either unicast or broadcast in nature. Many applications rely on Ethernet broadcasting to publish required information. For example the SMB protocol relies on broadcasts to publish information about specific file shares.

Broadcast Ethernet consists of all bits toggle on. In the following example, the tethereal command captures just broadcast traffic and picks up an SMB enabled device broadcasting over Ethernet:

```
# tethereal –nVi eth0 broadcast
Frame 2 (244 bytes on wire, 244 bytes captured)
    Arrival Time: Jul 13, 2008 17:09:20.645712000
    [Time delta from previous captured frame: 3.805290000 seconds]
    [Time delta from previous displayed frame: 3.805290000 seconds]
    [Time since reference or first frame: 3.805290000 seconds]
    Frame Number: 2
    Frame Length: 244 bytes
    Capture Length: 244 bytes
    [Frame is marked: False]
    [Protocols in frame: eth:ip:udp:nbdgm:smb:browser]
Ethernet II, Src: 00:c0:02:de:89:ef (00:c0:02:de:89:ef), Dst: ff:ff:ff:ff:ff:ff
(ff:ff:ff:ff:ff:ff)
    Destination: ff:ff:ff:ff:ff:ff (ff:ff:ff:ff:ff:ff)
        Address: ff:ff:ff:ff:ff:ff (ff:ff:ff:ff:ff:ff)
        .... ...1 .... .... .... .... = IG bit: Group address
(multicast/broadcast)
        .... ..1. .... .... .... .... = LG bit: Locally administered address (this
is NOT the factory default)
    Source: 00:c0:02:de:89:ef (00:c0:02:de:89:ef)
        Address: 00:c0:02:de:89:ef (00:c0:02:de:89:ef)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
        .... ..0. .... .... .... .... = LG bit: Globally unique address (factory
default)
    Type: IP (0x0800)
    Trailer: 00
```

The first 3 sets of octets in an Ethernet address refer to the “Organizational Identifier” or OID. The OID specifies the maker of the NIC card. The remaining 3 octets provide the unique address of the NIC card.

When used on the local LAN, the nmap utility discerns both the vendor of the NIC card (based on the OID). The following example probes the open UDP port from the previous example:

```
# nmap -sU -p U:138 192.168.1.200

Starting Nmap 4.11 ( http://www.insecure.org/nmap/ ) at 2008-07-13 17:44 CDT
Interesting ports on 192.168.1.200:
PORT    STATE         SERVICE
138/udp open|filtered netbios-dgm
MAC Address: 00:C0:02:DE:89:EF (Sercomm)

Nmap finished: 1 IP address (1 host up) scanned in 13.402 seconds
```

After some additional research on the Sercomm website, it appears that this NIC is part of an OEM embedded print server.

Alternatively, `nmap` supports a -O option that attempts to guess the OS. Embedded devices rarely open enough ports for nmap to accurately detect the OS.

4.1 Ethernet and the ARP Protocol

In order for packets to locate a host on a network of leave a TCP/IP network, they must have an IP address. Due to the reusable nature of IPv4 addresses, an IP address may be assigned to multiple systems over time. As a result, the "binding" of Ethernet addresses to IP addresses must be dynamic.

The Address Resolution Protocol (ARP) maintains the dynamic mapping of an Ethernet address to an IP address. All TCP/IP based systems use the ARP protocol to discover the latest mappings. The protocol stores these in a dynamic kernel based cache.

Systems use ARP over broadcast Ethernet protocol to discover mappings. There are 3 specific ways in which hosts use ARP to communicate:

- **Solicited ARP** – the most common form of ARP in which a node sends an ARP request over broadcast Ethernet looking for the Ethernet address of a specific node
- **Gratuitous ARP** – a node broadcasts it's own mapping over the Ethernet to inform systems that it has acquired a new IP address, commonly used when an OS brings a NIC online after a reboot
- **Proxy ARP** – a node (usually a router) replies to ARP requests on behalf of another node

The ARP protocol is unauthenticated and therefore open to abuse. Almost all "man-in-the-middle" network attacks start with corrupting (poisoning) ARP caches.

All Linux systems provide the `arp` command to check network connectivity. The `arp` command displays the Ethernet address to IP mappings for a system. These mappings are dynamic and will update and flush over time.

The following example commands demonstrate how to check to the default gateway for a system (`192.168.1.1`) and then to make sure an ARP entry exists in the ARP table:

```
# netstat -rn
Kernel IP routing table
Destination  Gateway       Genmask         Flags   MSS Window  irtt Iface
192.168.1.0  0.0.0.0       255.255.255.0   U         0 0          0 eth0
169.254.0.0  0.0.0.0       255.255.0.0     U         0 0          0 eth0
0.0.0.0     192.168.1.1     0.0.0.0        UG        0 0          0 eth0

# arp -an
? (192.168.1.102) at 00:13:CE:D8:23:0C [ether] on eth0
? (192.168.1.1) at <incomplete> on eth0
? (192.168.1.220) at 00:90:27:F6:0E:DA [ether] on eth0
```


In the previous entry, it appears that the ARP entry for the router has been flushed from the cache due to inactivity. The next time the host attempts to communicate with the router, it will first send an ARP request.

The following example capture shows an ARP in action. When attempting to use the ping command, the node (192.168.1.60) checks the ARP cache, realizes the router (192.168.1.1) entry is not there, and then sends out an ARP request.

```
# ping 192.168.1.1

# tcpdump -e -q -ni eth0 arp
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes

20:08:57.656098 00:0c:29:a3:84:72 > Broadcast, ARP, length 42: arp who-
has 192.168.1.1 tell 192.168.1.60

20:08:57.657638 00:06:25:77:63:8b > 00:0c:29:a3:84:72, ARP, length 60:
arp reply 192.168.1.1 is-at 00:06:25:77:63:8b

# arp -an
? (192.168.1.102) at 00:13:CE:D8:23:0C [ether] on eth0
? (192.168.1.1) at 00:06:25:77:63:8B [ether] on eth0
? (192.168.1.220) at 00:90:27:F6:0E:DA [ether] on eth0
```

Alternately, the `tethereal` displays the entire header format of the ARP request and reply.

```
# tethereal -V -c 2 -ni eth0 arp
<snip>

Address Resolution Protocol (request)
    Hardware type: Ethernet (0x0001)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (0x0001)
    Sender MAC address: 00:0c:29:a3:84:72 (00:0c:29:a3:84:72)
    Sender IP address: 192.168.1.60 (192.168.1.60)
    Target MAC address: 00:00:00:00:00:00 (00:00:00:00:00:00)
    Target IP address: 192.168.1.1 (192.168.1.1)
<snip>

Address Resolution Protocol (reply)
    Hardware type: Ethernet (0x0001)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (0x0002)
    Sender MAC address: 00:06:25:77:63:8b (00:06:25:77:63:8b)
    Sender IP address: 192.168.1.1 (192.168.1.1)
    Target MAC address: 00:0c:29:a3:84:72 (00:0c:29:a3:84:72)
    Target IP address: 192.168.1.60 (192.168.1.60)
```


In the previous output, the ARP request is sent to the broadcast Ethernet. The router, however, send the ARP reply back to the unicast address of the asking host (`192.168.1.60`).

The ARP protocol was designed to handle different iterations of request/response. The `Opcode` field defines the exact type of request. Many of the `Opcodes` have been deprecated over the years.

4.1.0 Case Study – Router Down or Ignoring You?

The most common problem with Internet connectivity is access to the upstream router. The quickest way to tell whether there it is an upstream connectivity problem or an access control issue is to examine the underlying ARP protocol.

The following example demonstrates a router (192.168.1.1) that is completely unavailable. The node (192.168.1.60) attempts a ping and receives a host unreachable:

```
# ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
From 192.168.1.60 icmp_seq=2 Destination Host Unreachable
From 192.168.1.60 icmp_seq=3 Destination Host Unreachable
From 192.168.1.60 icmp_seq=4 Destination Host Unreachable
From 192.168.1.60 icmp_seq=5 Destination Host Unreachable
From 192.168.1.60 icmp_seq=6 Destination Host Unreachable
From 192.168.1.60 icmp_seq=7 Destination Host Unreachable
```

The corresponding ARP requests go unanswered.

```
# tcpdump -e -q -ni eth0 arp or icmp and host 192.168.1.60
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decodelistening on eth0, link-type EN10MB (Ethernet), capture size 96
bytes

20:33:23.758181 00:0c:29:a3:84:72 > Broadcast, ARP, length 42: arp who-
has 192.168.1.1 tell 192.168.1.60

20:33:25.759101 00:0c:29:a3:84:72 > Broadcast, ARP, length 42: arp who-
has 192.168.1.1 tell 192.168.1.60

20:33:26.758071 00:0c:29:a3:84:72 > Broadcast, ARP, length 42: arp who-
has 192.168.1.1 tell 192.168.1.60
```

In the next example, the router is blocking the IP address at the IP protocol. The node receives the same reply from the router:

```
# ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
From 192.168.1.60 icmp_seq=2 Destination Host Unreachable
From 192.168.1.60 icmp_seq=3 Destination Host Unreachable
From 192.168.1.60 icmp_seq=4 Destination Host Unreachable
From 192.168.1.60 icmp_seq=5 Destination Host Unreachable
From 192.168.1.60 icmp_seq=6 Destination Host Unreachable
```



```
From 192.168.1.60 icmp_seq=7 Destination Host Unreachable
```

However, the capture shows that the router replied to the ARP request, but is not answering the ping requests:

```
# tcpdump -q -ni eth0 arp or icmp and host 192.168.1.60
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decodelistening on eth0, link-type EN10MB (Ethernet), capture size 96
bytes

20:36:43.396475 arp who-has 192.168.1.1 tell 192.168.1.60
20:36:43.398559 arp reply 192.168.1.1 is-at 00:06:25:77:63:8b

20:36:43.398573 IP 192.168.1.60 > 192.168.1.1: ICMP echo request, id
13933, seq 1, length 64
20:36:44.395473 IP 192.168.1.60 > 192.168.1.1: ICMP echo request, id
13933, seq 2, length 64
20:36:45.395437 IP 192.168.1.60 > 192.168.1.1: ICMP echo request, id
13933, seq 3, length 64
20:36:46.396394 IP 192.168.1.60 > 192.168.1.1: ICMP echo request, id
13933, seq 4, length 64
20:36:47.397387 IP 192.168.1.60 > 192.168.1.1: ICMP echo request, id
13933, seq 5, length 64
```

4.1.1 Case Study – Duplicate IP Addresses on the Network

Another common problem on a large flat network is that two systems may have the same IP address. In this situation, access to the intended system will be sporadic based on which host replies first to the ARP request.

In the following example, a node performs a ping on a specific host (192.168.1.102). The packet capture reveals two replies to the ARP request.

```
box:~# ping 192.168.1.102
PING 192.168.1.102 (192.168.1.102) 56(84) bytes of data.
64 bytes from 192.168.1.102: icmp_seq=1 ttl=128 time=5.83 ms
```

However, a capture of ARP traffic shows that two replies were sent to the original request. The first reply was from the Windows host. The LINUX host's entry came after.

```
box:~# tcpdump –q ni eth0 arp
tcpdump: verbose output suppressed, use v or vv for full protocol
decode listening on eth1, linktype EN10MB (Ethernet), capture
size 96 bytes
22:04:35.074216 arp who-has 192.168.1.102 tell 192.168.1.105
22:04:35.078448 arp reply 192.168.1.102 is-at 00:40:f4:83:48:24
22:04:35.079562 arp reply 192.168.1.102 is-at 00:0f:1f:17:ab:a7
```

A check of the ARP cache shows that the first ARP reply populates the cache.

```
box:~# arp -a
targus (192.168.1.220) at 00:02:55:74:41:1B [ether] on eth0
? (192.168.1.1) at 00:06:25:77:63:8B [ether] on eth0
? (192.168.1.102) at 00:40:f4:83:48:24 [ether] on eth0
```


An attempt to use ssh to connect to the remote host fails because the source host is attempting to connect to a host that does not have SSH running (possibly a Windows desktop).

```
box:~# ssh 192.168.1.102
OpenSSH_3.8.1p1 Debian8.
sarge.4, OpenSSL 0.9.7e 25 Oct 2004
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: Connecting to 192.168.1.102 [192.168.1.102] port 22.
```

4.1.2 Another Host Stole Your IP

When a Linux based host brings a network interface online, it sends out 4 gratuitous ARP requests. It is both checking to make sure that no other host is using its IP address and letting all nodes on the network know that it is online and using the IP address.

In the following example, a Linux host attempts to plumb an interface with an IP address of 192.168.1.64.

```
# ifup eth0

# tcpdump -q -ni eth0 arp
tcpdump: verbose output suppressed, use -v or -vv for full
protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96
bytes
20:54:47.181793 arp who-has 192.168.1.64 (Broadcast) tell 0.0.0.0
20:54:48.203058 arp who-has 192.168.1.64 (Broadcast) tell 0.0.0.0
20:54:48.959375 arp who-has 192.168.1.64 (Broadcast) tell 0.0.0.0
20:54:49.939827 arp who-has 192.168.1.64 (Broadcast) tell 0.0.0.0
20:54:50.981964 arp reply 192.168.1.64 is-at 00:0c:29:79:1e:90
20:54:52.502562 arp who-has 192.168.1.64 (Broadcast) tell
192.168.1.64
```

The host sends 4 ARP requests to the network checking to make sure no other nodes have the IP address already. After no replies, the host sends both a gratuitous reply to itself and a request back to the broadcast. Both ARP packets are considered gratuitous because they include the IP/Ethernet address of the host in both the source and destination fields and broadcast to the Ethernet.

A tethereal capture reveals both gratuitous ARP reply and requests. The first example shows the host replying back to itself sent to the Ethernet broadcast:


```
# tethereal –nVi eth0 arp

<snip>
Ethernet II, Src: 00:0c:29:79:1e:90 (00:0c:29:79:1e:90), Dst:
ff:ff:ff:ff:ff:ff (ff:ff:ff:ff:ff:ff)

<snip>
Address Resolution Protocol (reply/gratuitous ARP)
    Hardware type: Ethernet (0x0001)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (0x0002)
    Sender MAC address: 00:0c:29:79:1e:90 (00:0c:29:79:1e:90)
    Sender IP address: 192.168.1.64 (192.168.1.64)
    Target MAC address: 00:0c:29:79:1e:90 (00:0c:29:79:1e:90)
    Target IP address: 192.168.1.64 (192.168.1.64)
```

The same goes for the gratuitous ARP request.

```
<snip>
Ethernet II, Src: 00:0c:29:79:1e:90 (00:0c:29:79:1e:90), Dst:
ff:ff:ff:ff:ff:ff (ff:ff:ff:ff:ff:ff)

<snip>
Address Resolution Protocol (request/gratuitous ARP)
    Hardware type: Ethernet (0x0001)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (0x0001)
    Sender MAC address: 00:0c:29:79:1e:90 (00:0c:29:79:1e:90)
    Sender IP address: 192.168.1.64 (192.168.1.64)
    Target MAC address: ff:ff:ff:ff:ff:ff (ff:ff:ff:ff:ff:ff)
    Target IP address: 192.168.1.64 (192.168.1.64)
```

If a node already has an IP address, the exchange is much shorter. In the following example, the Linux host sends out the ARP request and immediately gets a reply that the IP address of `192.168.1.64` is already taken.

```
# tcpdump -q -ni eth0 arp
tcpdump: verbose output suppressed, use -v or -vv for full
protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96
bytes
21:08:06.965101 arp who-has 192.168.1.64 (Broadcast) tell 0.0.0.0
21:08:06.965104 arp reply 192.168.1.64 is-at 00:0c:29:85:f5:6a
```


5.0 Troubleshooting IP Connectivity

For any system to be available on a network, it must have a valid IP address. This IP address must be placed on the correct network with the appropriate routing entry to traverse networks. In order to differentiate between the local IP network and external networks, a node must apply the appropriate subnet mask. The subnet mask determines both the network address and the broadcast address.

The following packet capture displays the format of an IP header:

```
Internet Protocol, Src: 192.168.1.200 (192.168.1.200), Dst: 192.168.1.220
(192.168.1.255)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 229
    Identification: 0x22ab (8875)
    Flags: 0x00
        0... = Reserved bit: Not set
        .0.. = Don't fragment: Not set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 30
    Protocol: UDP (0x11)
    Header checksum: 0xf445 [correct]
        [Good: True]
        [Bad : False]
    Source: 192.168.1.200 (192.168.1.200)
    Destination: 192.168.1.255 (192.168.1.220)
```

Important fields to point out in the IP header include:

- **Source** – the source IP address
- **Destination** – the destination IP address
- **Flags** – determine if the packet is fragmented
- **Protocol** – the next layer protocol

5.0.1 Case Study – Quick IP Routing Troubleshooting

When a client host on a LAN can't communicate with the outside world, it can be one of 4 issues:

- no network connectivity
- misconfigured `/etc/nsswitch.conf`
- misconfigured or nonexistent `/etc/resolv.conf` for DNS
- misconfigured or wrong gateway information

The first 3 issues can be solved by viewing files and checking physical links. There is no real way to tell if the gateway entry is truly routing packets. The following example demonstrates how to monitor whether the gateway is routing packets.

The client host is unable to reach a host on the Internet.

```
box:~# ping yahoo.com
ping: unknown host yahoo.com
```

The client has a gateway configured in the routing table. However, there is no way to tell whether the gateway is actually routing.

```
box:~# netstat -rn
Kernel IP routing table
Destination       Gateway      Genmask      Flags MSS Window irtt  Iface
192.168.1.0       0.0.0.0     255.255.255.0 U        0      0 0    eth1
0.0.0.0           192.168.1.220 0.0.0.0     UG       0      0 0    eth1
```

The following packet capture is taken from the router. The packets are coming from the source host of `192.168.1.105`, but the interface is NOT showing the return packet.

```
[root@router ~]# tcpdump –ni eth0 not port 22
tcpdump: verbose output suppressed, use v or vv for full protocol decode
listening on eth0, linktype EN10MB (Ethernet), capture size 96 bytes

09:01:39.063347 IP 192.168.1.105.32770 > 4.2.2.2.domain: 49762+ A?
yahoo.com. (27)

09:01:44.075062 IP 192.168.1.105.32771 > 4.2.2.1.domain: 49762+ A?
yahoo.com. (27)
```

From the client prospective, the router is on the network as it replies to a `ping` request.

```
box:~# ping 192.168.1.220
PING 192.168.1.220 (192.168.1.220) 56(84) bytes of data.
64 bytes from 192.168.1.220: icmp_seq=1 ttl=64 time=2.69 ms
64 bytes from 192.168.1.220: icmp_seq=2 ttl=64 time=2.95 ms
```

However, the client is not receiving any replies from it's DNS request. Packets are going to the router, however they are getting dropped.


```
box:~# tcpdump –ni eth1 not port 22
tcpdump: verbose output suppressed, use v or vv for full protocol decode
listening on eth1, linktype EN10MB (Ethernet), capture size 96 bytes
09:10:06.367238 IP 192.168.1.105.32772 > 4.2.2.2.53: 49099+ A? yahoo.com.
(27)
09:10:11.381598 IP 192.168.1.105.32773 > 4.2.2.1.53: 49099+ A? yahoo.com.
(27)
```

5.1 The ICMP Protocol

The Internet Control Messaging Protocol (ICMP) was developed to test such connectivity. There are a series of ICMP tests available to test if the remote IP address is indeed reachable. The ICMP protocol consists of a TYPE and a CODE within a TYPE.

The most common testing command is the `ping` command. This utility leverages simple ICMP request (`TYPE: 8 CODE: 0`) and reply (`TYPE: 0 CODE: 0`) to see if the networking stack on the other system can process and reply to a packet.

The ICMP header is extremely simple. The following example capture displays the ICMP TYPE and CODE of a ping exchange between two hosts `192.168.1.60` and `192.168.1.220`:

```
# ping 192.168.1.220
# tethereal –nVi eth0 icmp
<snip>

Source: 192.168.1.220 (192.168.1.220)
    Destination: 192.168.1.60 (192.168.1.60)

<snip>

Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0 ()
    Checksum: 0x7590 [correct]
    Identifier: 0xeb4f
    Sequence number: 2 (0x0002)
    Data (56 bytes)

<snip>
Source: 192.168.1.60 (192.168.1.60)
    Destination: 192.168.1.220 (192.168.1.220)

<snip>
Internet Control Message Protocol
    Type: 0 (Echo (ping) reply)
    Code: 0 ()
    Checksum: 0x7d90 [correct]
    Identifier: 0xeb4f
    Sequence number: 2 (0x0002)
    Data (56 bytes)
```


5.2.0 Case Study – Misconfigured Broadcast Address 1

The broadcast address is often overlooked when determining network problems. Some network communications rely on a properly configured broadcast address including: NetBios (SMB), NTP, RIPv1, and ICMP Broadcast pings.

The following example demonstrates a standard ICMP broadcast ping. All hosts on the correct broadcast address reply to the broadcast ping:

```
box:~# ifconfig eth1
eth1 Link encap:Ethernet HWaddr 00:06:53:E4:8D:B8
inet addr:192.168.1.105 Bcast:192.168.1.255 Mask:255.255.255.0

<snip>

box:~# ping –b 192.168.1.255
WARNING: pinging broadcast address
PING 192.168.1.255 (192.168.1.255) 56(84) bytes of data.
64 bytes from 192.168.1.105: icmp_seq=1 ttl=64 time=0.052 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=150 time=3.54 ms (DUP!)
64 bytes from 192.168.1.220: icmp_seq=1 ttl=64 time=4.43 ms
(DUP!)
```

If the host does not have the correct broadcast address, other hosts will not reply to the broadcast ping requests. This is due to the fact that when the hosts on the network see the broadcast request, they will treat it as if an host was attempting to contact an individual host.

The following output shows that the system `192.168.1.105` has a class A subnet (`192.0.0.0`) although the network is class C (`192.168.1.0`).

```
box:~# ifconfig -eth1
eth1 Link encap:Ethernet HWaddr 00:06:53:E4:8D:B8
inet addr:192.168.1.105 Bcast:192.255.255.255 Mask:255.0.0.0
```

A broadcast ping returns no replies from the network.

```
box:~# ping –b 192.168.1.255
PING 192.168.1.255 (192.168.1.255) 56(84) bytes of data.
From 192.168.1.105 icmp_seq=1 Destination Host Unreachable
```

A packet capture further confirms the problem. The correct broadcast is `192.168.1.255`, however since the source host's broadcast is `192.255.255.255`, the source host is mistakenly trying to ARP for `192.168.1.255`, thinking it is a real host. The source host will never receive a valid ARP reply as no host on the network can have a `.255` in its last octet.

```
box:~# tcpdump -ni eth1
tcpdump: verbose output suppressed, use –v or –vv for full
protocol decode listening on eth1, linktype EN10MB (Ethernet),
capture size 96 bytes
22:31:01.512750 arp who-has 192.168.1.255 tell 192.168.1.105
```



```
22:31:02.512665 arp who-has 192.168.1.255 tell 192.168.1.105
```

5.2.1 Case Study – Misconfigured Broadcast Address 2

The following case study demonstrates how a misconfigured broadcast address affects NetBios discovery. In this case, a Linux file server (owasso-01) serves as an SMB based file server using Samba for a Windows network. However, the file server has an incorrect broadcast address:

```
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:90:27:F6:0E:DA
          inet addr:192.168.1.220  Bcast:192.255.255.255
Mask:255.0.0.0
```

The address should be a class A (192.168.1.0), but it is configured to be 192.0.0.0.

The file server owasso-01 (192.168.1.220) sends NetBios name registration packets on the broadcast address:

```
# tethereal –ni eth0 broadcast
<snip>

49  21.032326 192.168.1.220 -> 192.255.255.255 NBNS Registration NB OWASSO-01<20>
50  21.032327 192.168.1.220 -> 192.255.255.255 NBNS Registration NB OWASSO-01<03>
51  21.032328 192.168.1.220 -> 192.255.255.255 NBNS Registration NB OWASSO-01<00>

<snip>
```

A Windows end user then attempts to search for the Owasso-01 system in the Windows run function which returns a “Network Path Not Found.”

```
\\owasso-01

The network path was not found.
```

The tethereal output shows that the Windows desktop (192.168.1.103) sending out name queries to the broadcast address on a different subnet (the correct subnet):

```
54  22.538547 192.168.1.103 -> 192.168.1.255 NBNS Name query NB OWASSO-01<00>
55  22.542016 192.168.1.103 -> 192.168.1.255 NBNS Name query NB OWASSO-01<20>
56  23.004235 192.168.1.103 -> 192.168.1.255 NBNS Name query NB OWASSO-01<00>
57  23.008259 192.168.1.103 -> 192.168.1.255 NBNS Name query NB OWASSO-01<20>
```

5.2.2 Case Study – Stubborn DHCP Client

The DHCP protocol is critical to enabling access to corporate networks. The DHCP protocol relies heavily on both the ARP and Ethernet broadcast to dynamically assign IP addresses to client systems. The following packet capture walks through the process of a DHCP session between a client host and a DHCP server.

The DHCP protocol runs UDP over ports 67 and 68, but relies primarily on Ethernet and IP broadcasting (unless used in relay mode).

The first DHCP packet sent by the client goes to the Ethernet broadcast address. This is called the `Discover` packet. Since the client does not have an IP address, it uses `0.0.0.0` as a place holder.

```
14:52:59.602105 IP 0.0.0.0.bootpc > 255.255.255.255.bootps:
BOOTP/DHCP, Request from 00:0c:29:79:1e:90 (oui Unknown), length
3000
```

The DHCP selects the next IP address (`192.168.1.104`) in the pool and sends out an ARP to make sure that address is not taken:

```
4.475738 00:06:25:77:63:8b > ff:ff:ff:ff:ff:ff ARP Who-has 192.168.1.104?
Tell 192.168.1.1
```

If the DHCP server does not get a reply, it then sends an OFFER to the DHCP client over the broadcast address.

```
14:53:00.000569 IP 192.168.1.103.bootps > 192.168.1.203.bootpc:
BOOTP/DHCP, Reply, length 300
```

The DHCP server confirms that the client has received the IP address by sending an ARP request with the new mapping and an ICMP ping.

```
14:53:01.661405 arp who-has 192.168.1.203 tell 192.168.1.103
14:53:01.701045 arp reply 192.168.1.203 is-at 00:0c:29:79:1e:90 (oui
Unknown)
14:53:01.701059 IP 192.168.1.103 > 192.168.1.203: ICMP echo request, id
14401, seq 0, length 28
14:53:01.702118 IP 192.168.1.203 > 192.168.1.103: ICMP echo reply, id
14401, seq 0, length 28
```

In the following case study, a Windows XP system reports a “Limited or no Connectivity” warning when an end user tries to boot a laptop on a DHCP enabled network. The following packet capture shows that the DHCP server is indeed still running and responding to requests. It appears that the client simply will not accept an address:

```
# tcpdump –ni eth0 port 67 and port 68

15:12:44.508899 IP 0.0.0.0.bootpc > 255.255.255.255.bootps: BOOTP/DHCP,
Request from 00:02:a5:df:8b:79, length 300

15:12:45.000571 IP 192.168.1.103.bootps > 192.168.1.204.bootpc:
BOOTP/DHCP, Reply, length 300

15:12:48.998491 IP 0.0.0.0.bootpc > 255.255.255.255.bootps: BOOTP/DHCP,
Request from 00:02:a5:df:8b:79, length 300
```



```
15:12:48.998881 IP 192.168.1.103.bootps > 192.168.1.204.bootpc:
BOOTP/DHCP, Reply, length 300

15:12:56.998832 IP 0.0.0.0.bootpc > 255.255.255.255.bootps: BOOTP/DHCP,
Request from 00:02:a5:df:8b:79, length 300

15:12:56.999405 IP 192.168.1.103.bootps > 192.168.1.204.bootpc:
BOOTP/DHCP, Reply, length 300

15:12:57.000764 IP 0.0.0.0.bootpc > 255.255.255.255.bootps: BOOTP/DHCP,
Request from 00:02:a5:df:8b:79, length 305

15:12:57.197904 IP 192.168.1.103.bootps > 192.168.1.204.bootpc:
BOOTP/DHCP, Reply, length 300
```


6.0 Troubleshooting TCP Connectivity

The TCP protocol binds an application service to a port. Whether or not that binding is available is determined by TCP. The protocol provides two way communications for applications between hosts on a network in the 3 following ways:

- **Connection Oriented** – uses series of flags to maintain the state of the connection
- **Reliability** – uses a packet sequencing algorithm to ensure both sides of a connection receive all packets
- **Stateful** – uses "windows" to optimize buffer sizes

When an application binds to a port, network applications may connect to the port. The nmap utility ships provides a rich set of tools to determine whether a port is open or not. The nmap example demonstrates how to check if port 25 is available on a remote system:

```
# nmap -p 25 192.168.1.220

Starting Nmap 4.00 ( http://www.insecure.org/nmap/ ) at 2006-10-
05 10:46 PDT
Interesting ports on 192.168.1.244:
PORT   STATE SERVICE
25/tcp open  smtp
MAC Address: 00:0C:29:9E:7F:84 (3com)

Nmap finished: 1 IP address (1 host up) scanned in 0.650 seconds
```

The important field to observe is the STATE field. There are three possible states for a given port:

- **open** – The port is open and accepting connections.
- **`closed`** – The port is certified closed because the remote server sent an RFC compliant reset.
- **`filtered`** – The port is probably firewalled because the remote server is available, but sent no reply.

The UDP protocol is the stateless counterpart to TCP. Due to its simplistic nature, the UDP protocol is not covered in this section.

6.1 TCP Connection Oriented

Each one of the TCP/IP flags specifies the state of the connection between two end nodes. These flags are embedded in the TCP/IP packet headers. They are a series of bits. A flag is set when the bit is turned on and not set when it is turned off. The following output from a verbose packet capture shows the TCP/IP packet headers:

```
# tethereal –nVi eth0

<snip>

Transmission Control Protocol, Src Port: 22 (22), Dst Port: 4095 (4095),
Seq: 10336, Ack: 0, Len: 1260
    Source port: 22 (22)
    Destination port: 4095 (4095)
    Sequence number: 10336    (relative sequence number)
    Next sequence number: 11596    (relative sequence number)
    Acknowledgement number: 0    (relative ack number)
    Header length: 20 bytes
    Flags: 0x0010 (ACK)
        0... .... = Congestion Window Reduced (CWR): Not set
        .0.. .... = ECN-Echo: Not set
        ..0. .... = Urgent: Not set
        ...1 .... = Acknowledgment: Set
        .... 0... = Push: Not set
        .... .0.. = Reset: Not set
        .... ..0. = Syn: Not set
        .... ...0 = Fin: Not set
    Window size: 15232
    Checksum: 0x4c3e [correct]
```

The flags are split into three logical groups:

- **Opening a connection** – Syn/Acknowledgement
- **Maintaining a connection** – Push/Ack
- **Closing a connection** – Fin/Ack/Reset

6.1.0 Opening a Connection

The SYN and ACK flags are responsible for opening a TCP/IP connection. In order for these flags to open a connection, they must initiate a "3 Way Handshake". The flags are passed in the following steps:

1. The source host sends a packet with the SYN flag set.
2. The destination host replies to that packet with an ACK to the source host SYN and additionally sends its own packet with a SYN flag.
3. The source host receives the ACK to its SYN and the SYN from the destination host. The source then replies back to the destination with an ACK flag to the destination's SYN flag.

The following tcpdump output displays a source (hosta) and destination (hostb) initiating a connection on the SMTP port:

```
# tetheral port 25
hosta -> hostb TCP 1445 > smtp [SYN] Seq=0 Len=0 MSS=1260
hostb -> hosta TCP smtp > 1445 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0
MSS=1460
hosta -> hostb TCP 1446 > smtp [ACK] Seq=1 Ack=1 Win=17640 Len=0
```

6.1.1 Closing a Connection

The TCP protocol closes a connection in two ways: either through clean (FIN/ACK) or through dirty (RST) ways. The clean version enables both hosts to agree to close a connection through a series of 4 packets.

The following example demonstrates a clean closure of a connection:

```
# tethereal –nVi eth0 port 80

<snip>

42.303610 192.168.1.60 -> 204.179.240.224 TCP 3292 > 80 [FIN, ACK] Seq=125
Ack=42280 Win=63760 Len=0 TSV=36026120 TSER=534105456

42.348614 204.179.240.224 -> 192.168.1.60 TCP 80 > 3292 [ACK] Seq=42280 Ack=126
Win=33180 Len=0 TSV=534105463 TSER=36026120

42.349215 204.179.240.224 -> 192.168.1.60 TCP 80 > 3292 [FIN, ACK] Seq=42280
Ack=126 Win=33180 Len=0 TSV=534105463 TSER=36026120

42.349238 192.168.1.60 -> 204.179.240.224 TCP 3292 > 80 [ACK] Seq=126 Ack=42281
Win=63760 Len=0 TSV=36026166 TSER=534105463
```

In the previous output, both sides of the connection send their own FIN packet and an ACK of the other's, closing the connection mutually.

The following demonstrates a dirty closure of a connection:

```
17.478174 192.168.1.220 -> 192.168.1.60 TCP 47252 > 25 [RST]
Seq=35 Win=0 Len=0
```

6.1.3 Case Study – Service Not Running

There are multiple reasons why a remote server may not respond to a client request. A common mistake is to assume that since a host is available at the IP level, it does not mean that it is available at the TCP level.

The host is available at the IP level as per the ping replies.

```
box:~# ping 192.168.1.220
PING 192.168.1.220 (192.168.1.220) 56(84) bytes of data.
64 bytes from 192.168.1.220: icmp_seq=1 ttl=64 time=2.71 ms
64 bytes from 192.168.1.220: icmp_seq=2 ttl=64 time=2.64 ms
```


The SMTP service is not available to the client.

```
box:~# telnet 192.168.1.220 25
Trying 192.168.1.220...
telnet: Unable to connect to remote host: Connection refused
```

Taking a look at the packet capture, it is clear that the mail service is not running. Standard TCP replies to closed ports is to send a RST flag to the source host.

```
box:~# tcpdump –ni eth1 port 25
tcpdump: verbose output suppressed, use v or vv for full protocol decode
listening on eth1, linktype EN10MB (Ethernet), capture size 96 bytes

09:18:12.495679 IP 192.168.1.105.32780 > 192.168.1.220.25: S
2557844136:2557844136(0) win 5840 <mss 1460,sackOK,timestamp 1643931
0,nop,wscale 0>

09:18:12.498346 IP 192.168.1.220.25 > 192.168.1.105.32780: R 0:0(0) ack
2557844137 win 0
```

6.1.4 Case Study – Service Denying Access

Some applications provide host based access control. TCP Wrappers have been around for quite some time. Their purpose is to do host based access control. Unlike closed ports, the port to the server is open. Upon connection to the server, a check is made to the `/etc/hosts.allow` and `/etc/hosts.deny` files. If the client is allowed to connect, then a standard TCP connection is made. If not, the server sends a TCP reset to the client. The following is an example of a TCP wrapped `ssh` service.

```
box:~# ssh 192.168.1.220
ssh_exchange_identification: Connection closed by remote host
```

The `ssh` server initiates a 3 way handshake with the client. Upon checking the deny status, the server then sends a clean termination of the connection.

```
box:~# tcpdump -ni eth1 port 22
tcpdump: verbose output suppressed, use v or vv for full protocol decode
listening on eth1, linktype EN10MB (Ethernet), capture size 96 bytes

10:24:44.508208 IP 192.168.1.105.32786 > 192.168.1.220.22: S
2304868053:2304868053(0) win 5840 <mss 1460,sackOK,timestamp 2043132
0,nop,wscale 0>

10:24:44.510770 IP 192.168.1.220.22 > 192.168.1.105.32786: S
762146323:762146323(0) ack 2304868054 win 5792 <mss 1460,sackOK,timestamp
44455276 2043132,nop,wscale 2>

10:24:44.510807 IP 192.168.1.105.32786 > 192.168.1.220.22: . ack 1 win
5840 <nop,nop,timestamp 2043132 44455276>

10:24:49.526296 IP 192.168.1.220.22 > 192.168.1.105.32786: F 1:1(0) ack 1
win 1448 <nop,nop,timestamp 44460292 2043132>

10:24:49.526647 IP 192.168.1.105.32786 > 192.168.1.220.22: F 1:1(0) ack 2
win 5840 <nop,nop,timestamp 2043634 44460292>
```



```
10:24:49.529124 IP 192.168.1.220.22 > 192.168.1.105.32786: . ack 2 win
1448 <nop,nop,timestamp 44460295 2043634>
```

6.1.5 Case Study – Firewall Blocking Access

When a port is blocked by a packet filter (IPTables or IPFilter for example), it may be open but filtered at the IP level. In this case, the client will send multiple SYN packets to the server and the server will not respond simply because the packet has been dropped by the filter.

```
box:~# telnet 192.168.1.220 25
Trying 192.168.1.220...
telnet: Unable to connect to remote host: No route to host
```

A look at the packet capture shows that the client sent 3 TCP SYN packets to the smtp port (25) that were simply dropped by the server and not replied to in the client.

```
box:~# tcpdump –ni eth1 port 25
tcpdump: verbose output suppressed, use v or vv for full protocol decode
listening on eth1, linktype EN10MB (Ethernet), capture size 96 bytes

10:35:02.302120 IP 192.168.1.105.32787 > 192.168.1.220.25: S
2934062463:2934062463(0) win 5840 <mss 1460,sackOK,timestamp 2104912
0,nop,wscale 0>

10:35:13.847729 IP 192.168.1.105.32788 > 192.168.1.220.25: S
2951328215:2951328215(0) win 5840 <mss 1460,sackOK,timestamp 2106066
0,nop,wscale 0>

10:35:19.518837 IP 192.168.1.105.32789 > 192.168.1.220.25: S
2947681121:2947681121(0) win 5840 <mss 1460,sackOK,timestamp 2106633
0,nop,wscale 0>
```

6.1.6 Case Study – Rude SMTP Server

In the following example, a client (192.168.1.220) has successfully connected to an SMTP server and has started an SMTP transaction. Halfway through that transaction, the server (192.168.1.60) sends a RESET to kill the connection.

```
# tethereal –ni eth0 port 25
8.160142 192.168.1.220 -> 192.168.1.60 SMTP Command: helo 0
8.160215 192.168.1.60 -> 192.168.1.220 TCP 25 > 47252 [ACK] Seq=40 Ack=9
Win=5792 Len=0 TSV=36961862 TSER=478843278
8.161051 192.168.1.60 -> 192.168.1.220 SMTP Response: 250 ng-
server.localhost
8.162012 192.168.1.220 -> 192.168.1.60 TCP 47252 > 25 [ACK] Seq=9 Ack=65
Win=5840 Len=0 TSV=478843282 TSER=36961863
17.455709 192.168.1.220 -> 192.168.1.60 SMTP Command: mail from:
foo@yahoo.com
17.456097 192.168.1.60 -> 192.168.1.220 TCP 25 > 47252 [RST] Seq=65 Win=0
Len=0
```


6.2 Connection Oriented - State Transitions

As an alternate to monitoring live connections, a network administrator may also use state transitions as a method of troubleshooting a connection. Throughout the entire connection, TCP states change as flags pass between two hosts on a connection.

The following states are relevant to connection troubleshooting:

- LISTEN – A network service on the local host is bound to a port and listening for inbound SYN connections (StrongMail MTA server on port 25 or StrongMail IQMP Server on port 9010, for example).
- SYN_SENT – The source host has attempted to initiate a connection with a remote host by sending the initial SYN packet, but the destination host has yet to send the ACK (StrongMail IQMP server attempts to connect to receiving MX and does not receive ACK, for example).
- SYN_RCVD – A remote host has sent the local server a SYN and the server has sent an ACK. However, the remote host has not sent the 3rd ack.
- ESTABLISHED – The source and destination hosts have successfully completed a three-way handshake (StrongMail IQMP server makes successful connection to MX server).
- CLOSE_WAIT – The local host has sent a FIN/ACK sequence to close its end, but the remote host has not sent its FIN/ACK.

The `netstat` provides flag information on the state of a connection between two hosts. By using `netstat`, an administrator may determine the state of a connection by interpreting the `netstat` "STATE" column. In the following example, a `netstat` command displays that the `postfix` MTA is bound on port 25:

```
# netstat -t -anp | egrep '25|State' | egrep 'State|LISTEN'
Proto Recv-Q Send-Q Local Address   Foreign Address   State    PID/Program name
tcp   0      0      0.0.0.0:25      0.0.0.0:*         LISTEN   1978/postfix
```


6.2.1 Case Study – Remote Host is Ignoring You

In the following example, a corporate MTA is attempting to send email to the yahoo.com domain. The netstat command shows that the receiving MX server is rather busy, responding to some connections and timing out othere. Some connections sit in the ESTABLISHED state while others sit in SYN_SENT

```
# netstat –t –a | egrep ‘State|yahoo.com’
Proto Recv-Q Send-Q Local Address    Foreign Address   State
tcp     0     1      example.com:47390   mx1.yahoo.com:25   SYN_SENT
tcp     0     1      example.com:47389   mx1.yahoo.com:25   SYN_SENT
tcp     0     1      example.com:47404   mx1.yahoo.com:25   SYN_SENT
tcp     0     1      example.com:46677   mx1.yahoo.com:25   SYN_SENT
tcp     0     1      example.com:46680   mx1.yahoo.com:25   SYN_SENT
tcp     0     1      example.com:46691   mx1.yahoo.com:25   SYN_SENT
tcp     0     1      example.com:46832   mx1.yahoo.com:25   ESTABLISHED
tcp     0     1      example.com:46833   mx1.yahoo.com:25   SYN_SENT
tcp     0     1      example.com:46634   mx1.yahoo.com:25   SYN_SENT
tcp     0     1      example.com:46901   mx1.yahoo.com:25   ESTABLISHED
```

6.2.2 Case Study – Network Performance or Denial of Service

In the following example, a local web server seems to be performing slowly. The netstat output reveals that there are multiple half open connections in the SYN_RCVD state. This means that the web server has responded to the ACK of the SYN/ACK, but has not received the final ACK from the remote host. This situation could either mean that the upstream routers are dropping packets or a malicious attacker is attempting to DoS the system.

```
# netstat -t -an | egrep 'State | SYN_RECV'
Proto Recv-Q Send-Q Local Address               Foreign Address             State
tcp        0      0 192.168.1.103:80            151.20.250.144:6743         SYN_RECV
tcp        0      0 192.168.1.103:80            139.60.115.170:2687         SYN_RECV
tcp        0      0 192.168.1.103:80            60.249.54.227:24917         SYN_RECV
tcp        0      0 192.168.1.103:80            218.77.61.69:48692          SYN_RECV
tcp        0      0 192.168.1.103:80            97.133.21.125:21599         SYN_RECV
tcp        0      0 192.168.1.103:80            44.53.252.60:26569          SYN_RECV
tcp        0      0 192.168.1.103:80            244.64.87.40:42981          SYN_RECV
tcp        0      0 192.168.1.103:80            180.108.72.188:39119        SYN_RECV
tcp        0      0 192.168.1.103:80            157.75.153.0:59138          SYN_RECV
```


6.3 TCP Reliability and Statefulness

In addition to flag, the TCP protocol uses a packet sequencing and acknowledgement algorithm (reliability) and a window sized buffer (state) to determine how many packets each side of the connection can handle before acknowledgement.

The following example `tcpdump` output displays both the packet sequencing and window advertisement (`win`).

```
16:15:58.342003 IP 192.168.1.60.ssh > 192.168.1.102.cecsvc: P
58032:58176(144) ack 641 win 19584
```

The TCP sequencing algorithm enables hosts to receive and assemble TCP packets in or out of order. In addition to this, the algorithm enables both sides of the connection to determine which packets are lost and selectively retransmit those packets (Selective Acknowledgement).

The TCP window buffer tells a receiving host how many packets that can be processed on its buffer. Once the client reaches the window buffer limit, it will send and acknowledgement to the sending host to continue the next round of TCP packets. Hosts constantly adjust TCP window sizes depending on network latency and system load.

Troubleshooting sequencing and window sizes is rather difficult using libpcap based protocols (including utilities like WireShark). As a result, this paper uses the utility `tcptrace` as a way to summarize information about a connection. This utility is described in section 8 of this paper.

7.0 Introducing Network Monitoring

Out of all the subsyetms to monitor, networking is the hardest to monitor. This is due primarily to the fact that the network is abstract. There are many factors that are beyond a system's control when it comes to monitoring and performance. These factors include latency, collisions, congestion and packet corruption to name a few.

This section focuses on how to check the performance of Ethernet, IP and TCP.

7.1 Ethernet Configuration Settings

Unless explicitly changed, all Ethernet networks are auto negotiated for speed. The benefit of this is largely historical when there were multiple devices on a network that could be different speeds and duplexes.

Most enterprise Ethernet networks run at either 100 or 1000BaseTX. Use `ethtool` to ensure that a specific system is synced at this speed.

In the following example, a system with a 100BaseTX card is running auto negotiated in 10BaseT.

```
# ethtool eth0
Settings for eth0:
        Supported ports: [ TP MII ]
        Supported link modes:   10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
        Supports auto-negotiation: Yes
        Advertised link modes:  10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
        Advertised auto-negotiation: Yes
        Speed: 10Mb/s
        Duplex: Half
        Port: MII
        PHYAD: 32
        Transceiver: internal
        Auto-negotiation: on
        Supports Wake-on: pumbg
        Wake-on: d
        Current message level: 0x00000007 (7)
        Link detected: yes
```


The following example demonstrates how to force this card into 100BaseTX:

```
# ethtool -s eth0 speed 100 duplex full autoneg off
# ethtool eth0
Settings for eth0:
        Supported ports: [ TP MII ]
        Supported link modes:   10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
        Supports auto-negotiation: Yes
        Advertised link modes:  10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
        Advertised auto-negotiation: No
        Speed: 100Mb/s
        Duplex: Full
        Port: MII
        PHYAD: 32
        Transceiver: internal
        Auto-negotiation: off
        Supports Wake-on: pumbg
        Wake-on: d
        Current message level: 0x00000007 (7)
        Link detected: yes
```

7.2 Monitoring Network Throughput

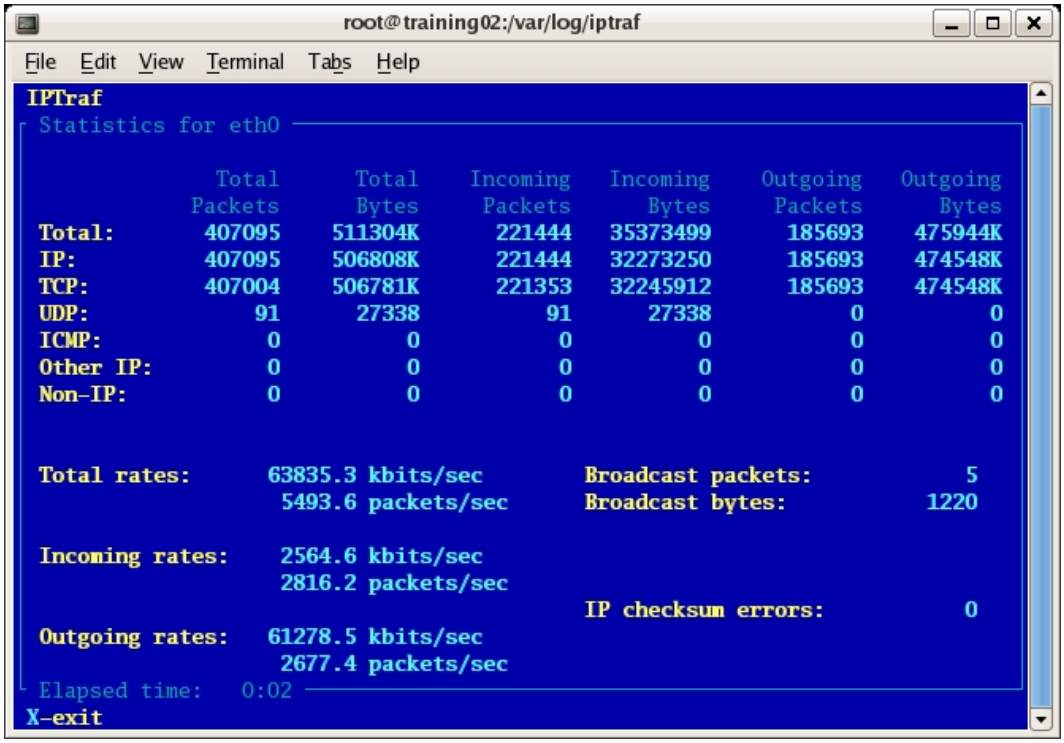
Just because an interface is now synchronized does not mean it is still having bandwidth problems. It is impossible to control or tune the switches, wires, and routers that sit in between two host systems. The best way to test network throughput is to send traffic between two systems and measure statistics like latency and speed.

7.2.0 Using iptraf

The `iptraf` utility (`http://iptraf.seul.org`) provides a dashboard of throughput per Ethernet interface.

```
# iptraf –d eth0
```


Figure 1: Monitoring for Network Throughput



The previous output shows that the system tested above is sending traffic at a rate of 61 mbps (7.65 megabytes). This is rather slow for a 100 mbps network.

7.2.1 Using `netperf`

Unlike `iptraf` which is a passive interface that monitors traffic, the netperf utility enables a system administrator to perform controlled tests of network throughput. This is extremely helpful in determining the throughput from a client workstation to a heavily utilized server such as a file or web server. The `netperf` utility runs in a client/server mode.

To perform a basic controlled throughput test, the `netperf` server must be running on the server system:

```
server# netserver
Starting netserver at port 12865
Starting netserver at hostname 0.0.0.0 port 12865 and family AF_UNSPEC
```

There are multiple tests that the `netperf` utility may perform. The most basic test is a standard throughput test. The following test initiated from the client performs a 30 second test of TCP based throughput:


```
client# netperf -H 192.168.1.230 -l 30
TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
192.168.1.230 (192.168.1.230) port 0 AF_INET
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec

 87380  16384  16384    30.02      89.46
```

The output shows that that the throughput on the network is around 89 mbps. This is exceptional performance for a 100 mbps network.

Another useful test using `netperf` monitors the amount of TCP request and response transactions taking place per second. The test accomplishes this by creating a single TCP connection and then sending multiple request/response sequences over that connection (`ack` packets back and forth with a byte size of 1). This behavior is similar to applications such as RDBMS executing multiple transactions or mail servers piping multiple messages over one connection.

The following example simulates TCP request/response over the duration of 30 seconds:

```
client# netperf -t TCP_RR -H 192.168.1.230 -l 30
TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET
to 192.168.1.230 (192.168.1.230) port 0 AF_INET
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

16384  87380  1        1       30.00    4453.80
16384  87380
```

In the previous output, the network supported a transaction rate of 4453 `psh/ack` per second using 1 byte payloads. This is somewhat unrealistic due to the fact that most requests, especially responses, are greater than 1 byte.

In a more realistic example, a `netperf` uses a default size of 2K for requests and 32K for responses:

```
client# netperf -t TCP_RR -H 192.168.1.230 -l 30 -- -r 2048,32768
TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
192.168.1.230 (192.168.1.230) port 0 AF_INET
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

16384  87380  2048     32768   30.00     222.37
16384  87380
```

The transaction rate reduces significantly to 222 transactions per second.

7.3 Monitoring for Error Conditions

The most common kind of error condition checked is for is packet collisions. Most enterprise networks are in a switched environment, practically eliminating collisions. However, with the increased usage of networked based services, there are other conditions that may arise. These conditions include dropped frames, backlogged buffers, and overutilized NIC cards.

Under extreme network loads, the `sar` command provides a report on all possible error types on a network.

```
# sar -n FULL  5 100
Linux 2.6.9-55.ELsmp (sapulpa)  06/23/2007

11:44:32 AM     IFACE   rxpck/s   txpck/s   rxbyt/s   txbyt/s   rxcmp/s   txcmp/s  rxmcst/s
11:44:37 AM        lo      6.00      6.00    424.40    424.40      0.00      0.00      0.00
11:44:37 AM      eth0      0.00      0.00      0.00      0.00      0.00      0.00      0.00
11:44:37 AM      sit0      0.00      0.00      0.00      0.00      0.00      0.00      0.00

11:44:32 AM     IFACE   rxerr/s   txerr/s    coll/s  rxdrop/s  txdrop/s  txcarr/s  rxfram/s  rxfifo/s  txfifo/s
11:44:37 AM        lo      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
11:44:37 AM      eth0      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
11:44:37 AM      sit0      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00

11:44:32 AM    totsck    tcpsck    udpsck    rawsck   ip-frag
11:44:37 AM       297        79         8         0         0
```

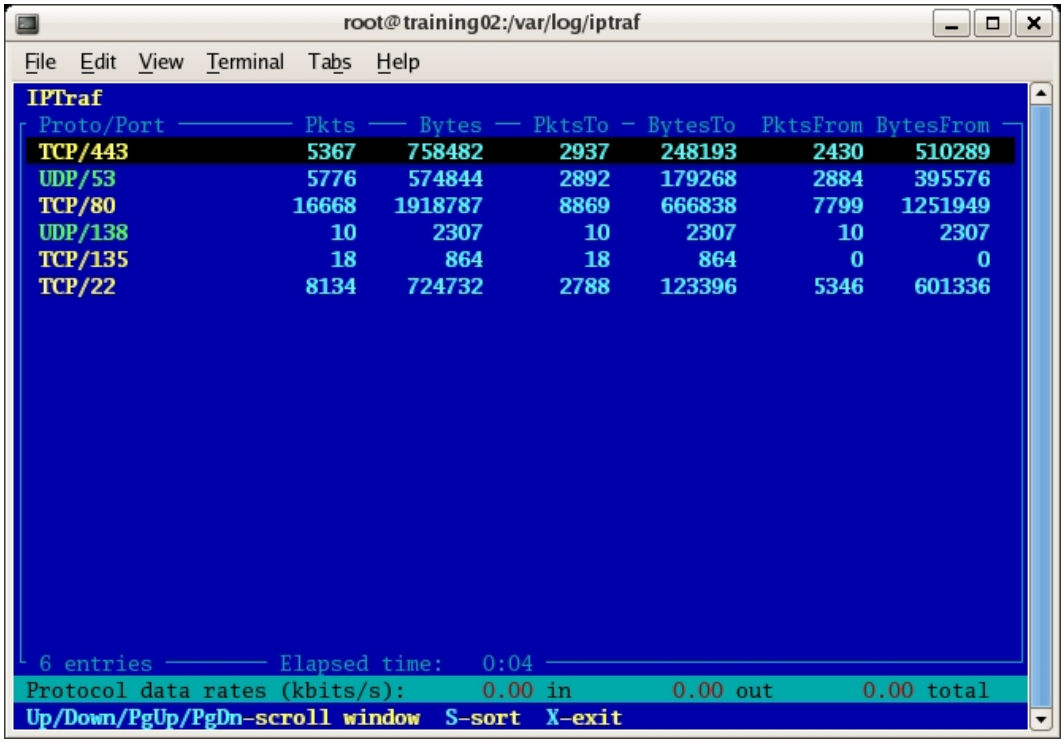
Table 3: Types of Network Errors

Field	Description
`rxerr/s`	rate of receive errors
`tcerr/s`	rate of transmit errors
`coll/s`	rate of collisions
`rxdrop/s`	received frames dropped due to kernel buffer shortage
`txdrop/s`	trasnmitted frames dropped due to kernel buffer shortage
`txcarr/s`	carrier errors
`rxfram/s`	frame alignement errors
`rxfifo/s`	receiving FIFO errors
`tcfifo/s`	trasmitted FIFO errors

7.4 Monitoring Traffic Types

Certain systems are designed to serve different traffic. For instance, a web server serves traffic over port 80 and a mail server over port 25. The `iptraf` tool displays the highest volume of traffic per TCP port.

Figure 2: Monitoring TCP Traffic per Port



7.5 Displaying Connection Statistics tcptrace

The tcptrace utility provides detailed TCP based information about specific connections. The utility uses libpcap based files to perform and an analysis of specific TCP sessions. The utility provides information that is sometimes difficult to catch in a TCP stream. This information includes:

- **TCP Retransmissions** – the amount of packets that needed to be sent again and the total data size
- **TCP Window Size**s – identify slow connections with small window sizes
- **Total throughput of the connection**
- **Connection duration**

7.5.0 Case Study – Using tcptrace

The tcptrace utility may be available in some Linux software repositories. This paper uses a precompiled package from the following website: http://dag.wieers.com/rpm/packages/tcptrace. The tcptrace command takes a source libpcap based file as an input. Without any options, the utility lists all of the unique connections captured in the file.

The following example uses a libpcap based input file called bigstuff:

```
# tcptrace bigstuff
1 arg remaining, starting with 'bigstuff'
Ostermann's tcptrace -- version 6.6.7 -- Thu Nov  4, 2004

146108 packets seen, 145992 TCP packets traced
elapsed wallclock time: 0:00:01.634065, 89413 pkts/sec analyzed
trace file elapsed time: 0:09:20.358860
TCP connection info:
  1: 192.168.1.60:pcanywherestat - 192.168.1.102:2571 (a2b)       404>  450<
  2: 192.168.1.60:3356 - ftp.strongmail.net:21 (c2d)               35>   21<
  3: 192.168.1.60:3825 - ftp.strongmail.net:65023 (e2f)             5>    4<
(complete)
  4: 192.168.1.102:1339 - 205.188.8.194:5190 (g2h)                  6>    6<
  5: 192.168.1.102:1490 - cs127.msg.mud.yahoo.com:5050 (i2j)        5>    5<
  6: py-in-f111.google.com:993 - 192.168.1.102:3785 (k2l)          13>   14<

<snip>
```

In the previous output, each connection has a number associated with it and the source and destination host. The most common option to tcptrace is the –l and –o option which provide detailed statistics on a specific connection.

The following example lists all of the statistics for connection #16 in the `bigstuff` file:

```
# tcptrace -l -o1 bigstuff
1 arg remaining, starting with 'bigstuff'
Ostermann's tcptrace -- version 6.6.7 -- Thu Nov  4, 2004

146108 packets seen, 145992 TCP packets traced
elapsed wallclock time: 0:00:00.529361, 276008 pkts/sec analyzed
trace file elapsed time: 0:09:20.358860
TCP connection info:
32 TCP connections traced:
TCP connection 1:
        host a:        192.168.1.60:pcanywherestat
        host b:        192.168.1.102:2571
        complete conn: no       (SYNs: 0)  (FINs: 0)
        first packet:  Sun Jul 20 15:58:05.472983 2008
        last packet:   Sun Jul 20 16:00:04.564716 2008
        elapsed time:  0:01:59.091733
        total packets: 854
        filename:      bigstuff
   a->b:                              b->a:
     total packets:           404           total packets:           450
     ack pkts sent:           404           ack pkts sent:           450
     pure acks sent:           13           pure acks sent:          320
     sack pkts sent:            0           sack pkts sent:            0
     dsack pkts sent:           0           dsack pkts sent:           0
     max sack blks/ack:         0           max sack blks/ack:         0
     unique bytes sent:     52608           unique bytes sent:     10624
     actual data pkts:        391           actual data pkts:        130
     actual data bytes:     52608           actual data bytes:     10624
     rexmt data pkts:           0           rexmt data pkts:           0
     rexmt data bytes:          0           rexmt data bytes:          0
     zwnd probe pkts:           0           zwnd probe pkts:           0
     zwnd probe bytes:          0           zwnd probe bytes:          0
     outoforder pkts:           0           outoforder pkts:           0
     pushed data pkts:        391           pushed data pkts:        130
     SYN/FIN pkts sent:       0/0           SYN/FIN pkts sent:       0/0
     urgent data pkts:          0 pkts      urgent data pkts:          0 pkts
     urgent data bytes:         0 bytes     urgent data bytes:         0 bytes
     mss requested:             0 bytes     mss requested:             0 bytes
     max segm size:           560 bytes     max segm size:           176 bytes
     min segm size:            48 bytes     min segm size:            80 bytes
     avg segm size:           134 bytes     avg segm size:            81 bytes
     max win adv:           19584 bytes     max win adv:           65535 bytes
     min win adv:           19584 bytes     min win adv:           64287 bytes
     zero win adv:              0 times     zero win adv:              0 times
     avg win adv:           19584 bytes     avg win adv:           64949 bytes
     initial window:          160 bytes     initial window:            0 bytes
     initial window:            2 pkts      initial window:            0 pkts
     ttl stream length:        NA           ttl stream length:        NA
     missed data:              NA           missed data:              NA
     truncated data:        36186 bytes     truncated data:         5164 bytes
     truncated packets:       391 pkts      truncated packets:       130 pkts
     data xmit time:      119.092 secs      data xmit time:      116.954 secs
     idletime max:       441267.1 ms        idletime max:       441506.3 ms
     throughput:              442 Bps       throughput:               89 Bps
```


7.5.1 Case Study – Calculating Retransmission Percentages

It is almost impossible to identify which connections have severe enough retransmission problems that require analysis. The `tcptrace` utility has the ability to use filters and Boolean expressions to locate problem connections. On a saturated network with multiple connections, it is possible that all connections may experience retransmissions. The key is to locate which ones are experiencing the most.

In the following example, the tcptrace command uses a filter to locate connections that retransmitted more than 100 segments:

```
# tcptrace -f'rexmit_segs>100' bigstuff
Output filter: ((c_rexmit_segs>100)OR(s_rexmit_segs>100))
1 arg remaining, starting with 'bigstuff'
Ostermann's tcptrace -- version 6.6.7 -- Thu Nov  4, 2004

146108 packets seen, 145992 TCP packets traced
elapsed wallclock time: 0:00:00.687788, 212431 pkts/sec analyzed
trace file elapsed time: 0:09:20.358860
TCP connection info:
 16: ftp.strongmail.net:65014 - 192.168.1.60:2158 (ae2af) 18695> 9817<
```

In the previous output, connection #16 experienced had more than 100 retransmissions. From here, the tcptrace utility provides statistics on just that connection:

```
# tcptrace -l -o16 bigstuff
 arg remaining, starting with 'bigstuff'
Ostermann's tcptrace -- version 6.6.7 -- Thu Nov  4, 2004

146108 packets seen, 145992 TCP packets traced
elapsed wallclock time: 0:00:01.355964, 107752 pkts/sec analyzed
trace file elapsed time: 0:09:20.358860
TCP connection info:
32 TCP connections traced:
==============================
TCP connection 16:
        host ae:       ftp.strongmail.net:65014
        host af:       192.168.1.60:2158
        complete conn: no       (SYNs: 0)  (FINs: 1)
        first packet:  Sun Jul 20 16:04:33.257606 2008
        last packet:   Sun Jul 20 16:07:22.317987 2008
        elapsed time:  0:02:49.060381
        total packets: 28512
        filename:      bigstuff
   ae->af:                           af->ae:

<snip>

     unique bytes sent:  25534744          unique bytes sent:         0
     actual data pkts:      18695          actual data pkts:          0
     actual data bytes:  25556632          actual data bytes:         0
     rexmt  data pkts:       1605          rexmt data pkts:           0
     rexmt  data bytes:   2188780          rexmt data bytes:          0
```


To calculate the retransmission rate:

```
rexmt/actual * 100 = Retransmission rate
```

Or

```
1605/18695* 100 = 8.5%
```

The previous connection had a retransmission rate of 8.5% which is the cause of the slow connection.

Appendix A - Troubleshooting DNS Issues

Improper DNS settings constitute a majority of the problems on StrongMail systems when attempting to send mail. All major ISPs require that any domain or IP address attempting to send mail have both forward and reverse DNS records. The domain must resolve both ways in order for a mail to be accepted.

There are many different domains that can be specified in an email message. It is considered best practice to make sure that all of these domains specified resolve to a PTR record in a DNS server. These domains in an email message include:

- **Sending domain in envelope header** – This is the default domain setup in a StrongMail system.
- **HELO Hostname** – This is the name the StrongMail server presents when initiating a data connection with a receiving MX server. This is configurable if Virtual Server Groups are enabled.
- **Return Path** – This is the domain in the user specified bounce address that is part of a mailing configuration file.

Using the nslookup Utility

The `nslookup` utility ships with both Linux and Microsoft Windows systems. It provides an interactive command line interface which allows you to perform multiple tasks on multiple servers from within one session.

The following steps describe how to debug DNS issues using DNS:

1. Launch the `nslookup` utility in interactive mode.

```
# nslookup
>
```

2. Find the authoritative server for the records you wish to search. Set the query to equal the authoritative name server for the domain. The local DNS server you may be using may have incorrect cached data in it.

```
> set q=NS
> yahoo.com
Server:         127.0.0.1
Address:        127.0.0.1#53

yahoo.com       nameserver = dns.yahoo.com.
```

3. Connect to the authoritative server.

```
> server dns.yahoo.com
Default server: dns.yahoo.com
Address: 192.168.1.222#53
```


4. Set the query to find A records for the domain.

```
> set q=a
> yahoo.com
Server:         dns.yahoo.com
Address:        192.168.1.222#53

Name:   yahoo.com
Address: 192.168.1.232
```

5. Now, set the query to find PTR records for the domain.

```
> set q=ptr
> 192.168.1.232
Server:         dns.yahoo.com
Address:        192.168.1.222#53

232.1.168.192.in-addr.arpa      name = yahoo.com.
```

6. Finally check to see if an MX record exists for the domain.

```
> set query=mx
> yahoo.com
Server:         dns.yahoo.com
Address:        192.168.1.222#53

yahoo.com       mail exchanger = 0 yahoo.com.
```

7. Optionally, you can also set the query to check the Domain Key record for the domain. The TXT record type stands for “TEXT” and is a freeform record. The first part of the record is called the “selector” and you must know the selector name beforehand.

```
# nslookup
> set q=TXT
> mail1._domainkey.yahoo.com.
Server:         192.168.1.222
Address:        192.168.1.222#53

customer._domainkey.yahoo.com    text = "t=y\; k=rsa\;
p=MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC9thNIYshzgwsDCShKMHZb3qo
raTaue8YM6HQwa3U5m6Am52sOHLyKforKRkSgyvW+p+DrMoAFwhg1IrBnHUwcsnS8
AzrpmPgW8SsvKexai85PB6xttkOPKSR/UZ/bBonseMkzSwDnLsakmR9phyc2zEwst
+LnDvv8sD2CA8XU9wIDAQAB"
```


Appendix B – Linux Networking Files

There are multiple files used by RHEL to configure networking on a system. These files reside in the `/etc` filesystem. Each file handles different components of networking for a RHEL system. The files include:

- `/etc/sysconfig/network-scripts/ifcfg-eth*`
- `/etc/sysconfig/network`
- `/etc/hosts`
- `/etc/resolv.conf`
- `/etc/nsswitch.conf`
- `/etc/host.conf`
- `/etc/init.d/network`

Each of the following sections describes these network configuration files.

Configuring The `ifconfig-eth*` Files

The `ifconfig-eth*` files include the most critical networking information for an Ethernet interface. The system creates one of these files for each interface of the system. For example, if the system had an `eth0` and a virtual interface of `eth0:1`, the system have `ifconfig-eth0` and `ifconfig-eth0:1` files. The system reads these files at startup via the `/etc/rc3.d/S10network` run control script.

The file contains multiple fields that define critical networking information:

```
# cat /etc/sysconfig/network-scripts/ifcfg-eth0
DEVICE=eth0
BOOTPROTO=static
IPADDR=67.110.253.164
NETMASK=255.255.254.0
NETWORK=67.110.253.0
GATEWAY=67.110.253.1
ONBOOT=yes
TYPE=Ethernet
```

Each of the fields is described below:

- `DEVICE` – The logical device name of the Ethernet interface (`eth0, eth1, eth0:1`, etc.)
- `BOOTPROTO` – This can be set to either `static` or `DHCP`
- `IPADDR` – The IP address assigned to the specified `DEVICE`
- `NETMASK` – The subnet mask applied to the `IPADDR`
- `NETWORK` – The network number applied to the `IPADDR`

- GATEWAY – The IP address of the default router for the IPADDR
- ONBOOT – This can be set to either yes or no to enable or disable the DEVICE
- TYPE – This is the layer 2 media type (Ethernet, Token, ATM)

Configuring the network File

The network file determines whether or not to enable networking on the system. It also sets the system's physical node name.

```
# more /etc/sysconfig/network
NETWORKING=yes
HOSTNAME=cent
```

The physical node name differs from the hostname. The term "hostname" often refers to the name of the host for a specific IP address. Since IP addresses are often difficult to remember, a hostname is assigned to an IP address to make it easier to remember. If a system has two IP addresses, it may have two hostnames defined, but only one physical node name. Hostnames are defined in the /etc/hosts file.

Configuring the hosts File

The /etc/hosts file enables local IP address to hostname resolution on a system. This allows for a hostname instead of an IP address to be specified on the command line.

```
# more /etc/hosts
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1               localhost.localdomain localhost
192.168.65.165          cent example.com # us
192.168.65.167          eng01 # them

# ssh eng01
root@eng01's password:
```

Configuring the host.conf File

The /etc/host.conf file determines how the system resolves its own IP address. The options are either to use DNS or the local /etc/hosts file. In the following example, the system first checks the local /etc/hosts files first before resolving to DNS.

```
# more /etc/host.conf
order hosts,bind
```


Configuring the nsswitch.conf File

The /etc/nsswitch.conf file determines how the system looks up different types of system attributes including: usernames, passwords, groups, and time zones. In terms of networking, the nsswitch.conf file contains the hosts keyword. This keyword determines how the system resolves other host IP addresses. The options include: hosts file, DNS, LDAP, NIS, or NIS+.

RHEL ships with hosts and DNS as the options for hostname resolution:

```
# more /etc/resolv.conf

<snip>

#hosts:     db files nisplus nis dns
hosts:      files dns

<snip>
```

Configuring the resolv.conf File

The /etc/resolv.conf file defines the DNS servers and domains that the system uses when attempting to resolve hostnames. At a minimum, this file must define one IP address of one DNS server in order to provide DNS resolution to the system. Multiple DNS servers may be specified. The system uses the other entries based on a timeout policy. If the first defined DNS server does not respond, then the system tries the other DNS servers defined until one of the servers responds.

```
# cat /etc/resolv.conf
search example.com
nameserver 192.168.65.220
```

The search keyword is optional. This keyword defines a domain to append to a non fully quailified domain name (FQDN). In the following example, the user specifies both the FQDN and non-FQDN for a ping command.

```
# ping ns1.example.com
PING ns1.example.com (66.94.234.13) 56(84) bytes of data.
64 bytes from ns1.example.com (66.94.234.13): icmp_seq=0 ttl=52
time=27.1 ms

# ping ns1
PING ns1.example.com (66.94.234.13) 56(84) bytes of data.
64 bytes from ns1.example.com (66.94.234.13): icmp_seq=0 ttl=52
time=27.1 ms
```

Configuring the network Run Control Script

The /etc/init.d/network script reads all of the above mentioned configuration files, extracts all of the networking information, and configures the network interfaces on the system.

Any time a system administrator changes networking configuration files, it is recommended that the `network` script run to reflect the networking changes.

```
# service network stop
# service network start
```


Appendix B - Manual Network Configuration

RHEL provides a series of command utilities to configure network interfaces manually. Proper configuration of the networking files should be sufficient to enable networking on the system. In the event, however, that a temporary change to the network is necessary, these tools are available. Any manual network configuration does not survive a reboot.

Using the `ifconfig` Command

The `ifconfig` command both checks and sets the critical networking information needed for a system to function. This includes the IP address, netmask, broadcast, and network number settings.

In the following example, the `ifconfig` utility checks the settings for all Ethernet interfaces:

```
# ifconfig -a
eth0   Link encap:Ethernet  HWaddr 00:0C:29:4A:8A:E4
       inet addr:192.168.65.201  Bcast:192.168.65.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:918 errors:0 dropped:0 overruns:0 frame:0
          TX packets:96 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:141797 (138.4 Kb)  TX bytes:9533 (9.3 Kb)
          Interrupt:10 Base address:0x1400

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:17728 errors:0 dropped:0 overruns:0 frame:0
          TX packets:17728 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1265424 (1.2 Mb)  TX bytes:1265424 (1.2 Mb)
```

If you want to check a specific interface, include the interface name after typing the `ifconfig` command:

```
# ifconfig eth0
```

To create a new virtual network interface on the system called `eth0:1`, type the following `ifconfig` command to first enable the interface:

```
# ifconfig eth0:1 192.168.1.91 netmask 255.255.255.0 broadcast \
> 192.168.1.255 up
# ifconfig eth0:1
eth0:1    Link encap:Ethernet  HWaddr 00:90:27:F6:0E:D8
          inet addr:192.168.1.91  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:5 Base address:0x1060 Memory:fa104000-fa104038
```


In order to make this virtual interface available across reboots, a configuration file called `ifcfg-eth0:1` must be created in the `/etc/sysconfig/network-scripts` directory.

Using the `route` Command

The `route` command manipulates the system routing table. The system routing table is stored in memory by the kernel and it is dynamically built at system initialization. Like the `ifconfig` command, using the `route` command on the command line only makes a temporary change to the routing table.

In the following example, the `netstat` command is used to check to see if a default route exists on the system:

```
# netstat -rn
Kernel IP routing table
Destination  Gateway      Genmask         Flags   MSS Window  irtt Iface
192.168.1.0  0.0.0.0      255.255.255.0   U         0 0          0 eth0
169.254.0.0  0.0.0.0      255.255.0.0     U         0 0          0 eth0
```

Only the LAN route is specified in the routing table (`192.168.1.0`). This will only enable the system to send traffic to systems on the local LAN. In order to send traffic beyond the LAN to other IP networks and the Internet, the `route` command places a default gateway (`default gw`) in the routing table.

```
# route add default gw 192.168.168.1.1
# netstat -rn
Kernel IP routing table
Destination  Gateway       Genmask         Flags   MSS Window  irtt Iface
192.168.1.0  0.0.0.0       255.255.255.0   U         0 0          0 eth0
169.254.0.0  0.0.0.0       255.255.0.0     U         0 0          0 eth0
0.0.0.0     192.168.1.1     0.0.0.0        UG        0 0          0 eth0
```

The `Flags` field of the last entry in the routing table contains a `UG` for "Up and Gateway".

Using the `netconfig` Command

The `netconfig` command is a RHEL CLI utility that automates network configuration on a system. This command provides a simpler interface to network configuration and preserves the changes across reboots by writing the changes to the appropriate files.

To run the `netconfig` command, type:

```
# netconfig
```


Once you have completed the settings, select OK. The settings will not take effect until the system reboots or services are restarted on the command line. To restart services on the command line, run the following command:

```
# service network restart
```

Stopping and Starting Network Services

There are many different ways to stop and start network services on a system. The network run control script has already been mentioned in this guide. It is the recommended method for stopping and starting services. In the case of multiple interfaces, however, this method may not be the best. If multiple physical interfaces exist on the system (eth0 and eth1 for example) and both interfaces may not be offline at the same time, the `ifup` and `ifdown` commands may be used per specific interface.

The `ifup` and `ifdown` commands are shell scripts that read the `/etc/sysconfig/network-scripts/ifcfg-eth*` files and configure the network interfaces accordingly.

In the following example, the `ifup` and `ifdown` commands take `eth0` offline, but not `eth1`. First, verify that both interfaces are up:

```
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:B0:D0:DE:2F:1C
          inet addr:192.168.1.67  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::2b0:d0ff:fede:2f1c/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:699 errors:0 dropped:0 overruns:1 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:1
          collisions:0 txqueuelen:1000
          RX bytes:223496 (218.2 KiB)  TX bytes:438 (438.0 b)
          Interrupt:5 Base address:0xec00

# ifconfig eth1
eth1   Link encap:Ethernet  HWaddr 00:A0:C9:9C:E1:C5
```



```
          inet addr:67.110.253.161  Bcast:67.110.253.255  Mask:255.255.254.0
          inet6 addr: fe80::2a0:c9ff:fe9c:e1c5/64 Scope:Link
             UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
             RX packets:66976202 errors:0 dropped:0 overruns:0 frame:0
             TX packets:41625913 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:1000
             RX bytes:3212552412 (2.9 GiB)  TX bytes:3992947992 (3.7 GiB)
```

Using the `ifdown`, down the `eth0` interface:

```
# ifdown eth0
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:B0:D0:DE:2F:1C
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:759 errors:0 dropped:0 overruns:1 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:1
          collisions:0 txqueuelen:1000
          RX bytes:242746 (237.0 KiB)  TX bytes:438 (438.0 b)
          Interrupt:5 Base address:0xec00
```

Using the `ifup` command, up the `eth0` interface:

```
# ifup eth0
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:B0:D0:DE:2F:1C
          inet addr:192.168.1.67  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::2b0:d0ff:fede:2f1c/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:761 errors:0 dropped:0 overruns:1 frame:0
          TX packets:14 errors:0 dropped:0 overruns:0 carrier:1
          collisions:0 txqueuelen:1000
          RX bytes:243170 (237.4 KiB)  TX bytes:986 (986.0 b)
          Interrupt:5 Base address:0xec00
```

Using the ping Command:

The `ping` command is a simple test to check remote connectivity. It sends an IP packet with nothing more than a "Request" flag in it. If the remote server receives the packet, it will then send a "Reply" back to the system.

When troubleshooting remote connectivity, always start by testing connectivity to the default router:

```
# ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=150 time=3.73 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=150 time=3.20 ms

--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1003ms
rtt min/avg/max/mdev = 3.209/3.474/3.739/0.265 ms
```

If the router responds, then attempt to contact the remote host:


```
# ping ufsdump.org
PING 63.172.36.208 56(84) bytes of data.
64 bytes from 63.172.36.208: icmp_seq=1 ttl=64 time=6.07 ms
64 bytes from 63.172.36.208: icmp_seq=2 ttl=64 time=3.86 ms
```

If the ping command does not return ANY data (not even first line), then check DNS, as the domain name you are trying to contact does not resolve.

NOTE: ICMP test utilities only verify the IP address of a remote system, they do not provide any information on whether or not the TCP/UDP ports on a system are available. Thus, a successful ICMP test using the ping command does NOT mean that a service such as a web server is functioning.

If the domain does not respond to the ping, then attempt to ping any host on the Internet. This will determine if there is a connectivity issues on your end or an issue on their end. If you can ping any Internet system besides theirs, then it is an issue on their end.

```
# ping 4.2.2.1
PING 4.2.2.1 (4.2.2.1) 56(84) bytes of data.
64 bytes from 4.2.2.1: icmp_seq=1 ttl=246 time=74.5 ms
64 bytes from 4.2.2.1: icmp_seq=2 ttl=246 time=66.8 ms
64 bytes from 4.2.2.1: icmp_seq=3 ttl=246 time=67.3 ms

--- 4.2.2.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 66.874/69.604/74.557/3.514 ms
```


References

Blum, Richard. *Network Performance Open Source Toolkit.* Indianapolis, IN: Wiley Publishing, 2003

Burns, Kevin. *TCP/IP Analysis and Troubleshooting Toolkit*. Indianapolis, IN: Wiley Publishing, 2003

Haugdahl, J. Scott. *Network Analysis and Troubleshooting*. New York, NY: Addison-Wesly, 2000

Hoch, Darren. *Troubleshooting TCP/IP 2005*. http://www.ufsdump.org

Wieers, Dag. *DAG RPM Repository*. http://dag.wieers.com

