Universal Ping

This document describes **uping**, a universal ping utility that automatically switches between **ping** and **ping6** and supports TCP-layer pings.

1. Problem Statement

When a typical user wants to determine if a computer is up they'll probably run **ping hostname** to see if there's a response - some versions of **ping** are so simple they even report that the hostname is alive rather than the details of the ping request. But sometimes this isn't possible, practical, or just gives misleading information.

Anyone who's dealt with IPv6 on Linux has likely been very happy with the robustness of the support and frustrated at the need for separate ping and ping6 commands. I've created a script called **uping** which seeks to eliminate these issues by using **dig** to determine whether the host in question is IPv6 enabled.

Once you get rid of the IPv4/IPv6 issue you run into ICMP-based issues. First, ICMP is generally considered to be low priority traffic so it's more likely to be dropped or delayed in transit. This makes it less than ideal for a lot of things.

Often systems block ICMP traffic. ICMP has been known to cause a lot of issues (see "ping of death") because it's designed to be dumb so many implementations are pretty dumb. As such you can frequently find exploits involving malformed or unexpected ICMP packets. When I was a young whippersnapper a few Linux boxes could crash a school mainframe by sending large, appropriately formed ping packets. Today a few thousand computers can cripple a webserver with the same sort of attack.

Different services often go to different machines. In my home network a ping would be responded to by my router. A request for an SSH session would go to my server, a request for web traffic to a virtual machine - a successful ping means pretty much nothing.

An ICMP echo also can't determine if a service is up, just because you can ping a machine doesn't mean the media streaming is up.

2. IPv4 vs. IPv6

The main problem I wanted to solve is to automatically select IPv6 whenever possible, with an automatic

fallback to IPv4 for the unenlightened.

2.1. Using ping/ping6

The original scope of **uping** was to enable a single command to autodetect if a hostname was IPv6-enabled. Obviously you can manually work around this by using **ping6** and, if there is no response, fall back to **ping**. The issue with this is that it doesn't let people know how many IPv6 hosts are out there, most people will simply run **ping** and never know what they're missing.

3. ICMP vs. TCP

The second issue that crops up for me is the ability to ping a TCP port rather than just a dumb ICMP echo.

3.1. Using NMAP

NMAP is a very robust port scanner and can give you pretty easy liveness information on a specific port. It's a little like swatting a fly with a sledgehammer, but you can do a basic liveness check easily and a little scripting can get you a nice ping replacement. NMAP is generally useful and you can install it under Ubuntu by running **sudo apt-get install nmap**.

The basic use of a ping client is to determine liveness and latency, preferably with as little overhead as possible (there are some legitimate uses for specific packet sizes, but by and large a tiny packet is going to get you the best results). In the TCP world this means completing a three-way handshake. We want NMAP to send a SYN packet, then the far end system will send us a SYN-ACK (an acknowledgment that it got the packet, it will only do this if the port is open and listening). Finally we send an ACK packet to indicate that the connection is available. We then want to be nice and send a RST packet to indicate that we're done with the connection so it can be reset.

A SYN flood is a type of Denial of Service (DoS) attack in which a bunch of SYNs are sent that suck up resources on the far end but the sender never completes the connection. The insidious part of this is that you can send the SYNs from random IP addresses since you don't care about getting the ACK back. We don't want our pings to look like a SYN flood.

To simulate the polite behavior with NMAP, you can run the following command **nmap -p <port_number> -PS -PN [-6] <hostname>**. The **-p** parameter tells it not to bother with scanning all of the ports, just use the one you care about. The **-PS** parameter means to use a SYN/ACK mechanism to determine liveness. The **-PN** parameter is often not needed, but it tells NMAP to ignore whether a ping responds. Since we obviously think ICMP is not an appropriate protocol for our liveness it couldn't hurt to have it set. Finally, you may want to consider using a **-6** to use IPv6.

We can take a closer look at what this command does by running it against a host and using **sudo tcpdump -n -vvv -i <interface> host <host_ip>** and then running the command. The **-n** parameter prevents the program from looking up IP addresses, **-vvv** puts it into verbose mode (why not?), **-i** is usually eth0 or eth1 (eth1 for wireless on most laptops) and the **host** parameter will throw out packets not going to the host IP. Note that you'll want to ensure that you don't have any other connections open to the host, if you're SSHed into the machine you'll see all of those packets as well with this command (but you may be able to filter them out as well).

What you should see is a SYN packet (a Flag of [S]) being sent to the host, a SYN ACK ([S.]) coming back, an ACK ([.]) completing the handshake, and then a RST ACK ([R.]) to terminate the connection. This is a clean implementation in that the connection should not hang around.

3.2. Using hping3

There is also a lighter weight method that can create custom packets called **hping**. It can be installed under Ubuntu using **sudo apt-get install hping3** and is nice in that it provides a continuous ping. The problem is that to simulate the three-way handshake you need to be root for some reason.

To perform this function, run the command **hping3 -p <port_number> -S <hostname>**. The relevant part is the **-S** which will generate the SYN packet and the output will look pretty much like the output of the ping command. The limitation is that it doesn't yet support IPv6 so you're a little hosed if you're running a dual stack.

To perform this function, run the command **hping3 -p <port_number> -S <hostname>**. The relevant part is the **-S** which will generate the SYN packet and the output will look pretty much like the output of the ping command. The limitation is that it doesn't yet support IPv6 so you're a little hosed if you're running a dual stack. Both the lack of IPv6 support and the need to be privileged to create a TCP ping made me drop **hping3** as a possible solution.

What's really going on this this command? Well, first it does a DNS lookup (obviously!) and asks for an A record. Then it sends a packet with a SYN flag set. It receives the SYN ACK from the far end (assuming it's up, of course), then it sends a RST. This is a fairly clean implementation and no half-open connections should stay around. However note that it's a bit different than NMAP in that it never finishes the three-way handshake. Technically this is a bit nicer in that the TCP connection doesn't take up resources, but it's also less standard and may create some concern if the guy at the other end of the server is paying attention (which he likely isn't).

3.3. Issues With These Methods

There are a few issues with these methods. First, they only exercise the TCP engine. If you're testing if a webserver is up, you'll probably want to use an HTTP-layer utility like **curl** because it's possible that the

port is open but the server has gone wonky. The problem is tools for other protocols like SSH may not be as robust.

Latency is a bit variable in my experience. The latency measured isn't the end-to-end latency, it also is the latency involved in creating a new TCP connection. This may explain some of it, the fact that the tools may not be optimized for measuring latency may also be a part of it. On a fairly stable connection I just measured between 37.4 and 426.6ms after 269 packets with an average of 240.6ms.

Despite the relative niceness of these implementations, they still are doing something that a real-world connection would never do - that is setting up a connection (either completely or mostly) and then terminating it without sending a packet. Advanced intrusion detection devices and firewalls can pick up on these anomalous connections and raise alerts. Just the idea of a single client creating a new connections about once a second is probably enough for this trigger.

4. The uping Script

This script should run on most bash-enabled systems, but was built for Ubuntu (http://www.ubuntu.com/) 10.04 (Lucid Lynx), 10.10 (Maverick Meerkat) and 11.10 (Oneiric Ocelot). The **uping** script may be downloaded here (uping.tgz) or you may use my Launchpad repository listed in Appendix A. Just install the repo and then run **sudo apt-get update && sudo apt-get install uping**. For the copy-and-paste types the program listing appears below.

4.1. uping v1

The initial version of **uping** only supported ICMP. I'm keeping it around just in case anyone has been using it and doesn't want to modify their scripting and because it has the interesting property in that it will pass parameters off to **ping**/**ping6**, since **uping** version 2 uses both ICMP and TCP this becomes a little less doable. Skip past listing.

Example 1. uping1

```
#!/bin/bash
if [ ! -n "$1" ]; then
  echo Usage: $0 [ping_parameters] destination
  exit 1
fi

eval dest=\$$#

dig +search $dest AAAA |grep AAAA |grep \: > /dev/null

if [ $? = 0 ]; then
  echo Executing ping6 $@
  ping6 $@
```

```
  ping6result=$?
  if [ $ping6result = 0 ]; then
    exit 0
  fi
  echo Found IPv6 address but ping6 failed with exit code $ping6result, do you have IPv6?
else
  ping6result=2
  echo Failed to resolve hostname on IPv6.
fi

dig +search $dest | grep A | grep '[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}' >


if [ $? = 0 ]; then
  echo Executing ping $@
  ping $@
  pingresult=$?
  exit $pingresult
fi

echo Failed to resolve hostname on IPv4.
exit $ping6result
```

When executed without parameters, **uping** will exit indicating the syntax. In theory any standard ping parameters should work just fine, but note that the same parameters will be passed to **ping6** and **ping**.

The last parameter is assumed to be the hostname. First I use **dig** to try to find the IPv6 AAAA record for the hostname. If this succeeds (**dig** returns an exit code of 0) then I execute **ping6**. If **ping6** returns a non-zero exit code or if I can't find a AAAA record I proceed to try IPv4.

I again use **dig** to try to find the IPv4 A record. If it works I try **ping**, if not I report that I've failed to resolve an IPv4 host.

For IPv4-only hosts I exit with the **ping** exit code. For IPv6-only hosts I exit with the **ping6** exit code. For dual mode hosts I'll exit with the **ping** exit code.

4.2. uping v2

The next generation of **uping** provides a fundamentally different interface. First, I don't rely on the continuous ping command since this isn't supported by NMAP. This does allow me to do a little better processing, now the output looks pretty much identical whether you're working on IPv6, IPv4, ICMP or UDP. It mean that I need to manually create an interpretation for parameters common to ping, but I've already created an interpreter for the count parameter (`-c 10` will send 10 ping attempts and then quit). **uping** is designed to be mostly a liveness check, a function it performs admirably.

Note that you'll need NMAP for this to work with TCP, to install simply run **sudo apt-get install nmap** under Ubuntu/Debian or download the deb package in Section 4 and the dependencies should be taken care of for you. Skip past listing.

Example 2. uping2

```
#!/bin/bash

# Known issues:
# Bug in calculating percent loss - comes out to be greater than 100%.

# Note that this may be a viable regex to check for an IPv6 address, but it's
# so much easier to just assume [] is used...

# /^\s*((([0-9A-Fa-f]{1,4}:){7}([0-9A-Fa-f]{1,4}|:))|(([0-9A-Fa-f]{1,4}:){6}(:[0-9A-Fa-f]{1,

programname=‘basename $0‘

if [ $# -eq 0 ]; then
  echo Usage\:
  echo "  $programname [parameters] hostname[:portnum]

hostname can be an actual hostname, an IP address, or an IPv6 address
Examples:
$programname www.ebower.com           : sends ICMP ECHO to www.ebower.com
$programname www.ebower.com:443       : makes TCP connections on port 443
$programname 192.168.1.1              : sends ICMP ECHO to 192.168.1.1
$programname [2001:4830:113c::80]     : sends ICMP ECHO to 2001:4830:113c::80
$programname [2001:4830:113c::80]:80  : makes TCP connections on port 80

Parameters include:
--arp     Issue ARP/NDISC requests, no RTT available (same as --ndisc)
-c n      Issues n ping requests and quits
--count n Same as -c n.
-i ethn   Uses network interface for ARP requests (with --arp/--ndisc only)
--interface ethn  Same as -i ethn
--ndisc   Issue ARP/NDISC requests, no RTT available (same as --arp)"

  exit 0
fi

mode=ICMP
ipmode=nodns
count=-1

function parse_parameters {
  c=1;
  while [ $c -lt $# ]; do
    parameter=${!c}
    case $parameter in
      "--arp")
        mode=ARP
      ;;
```

```
      "-c" | "--count")
        c=$(( $c + 1 ))
        count=${!c}
        if [ ! $(echo "$count" | grep "[^0-9]") = "" ]; then
          echo Expected integer for -c parameter, found $count.
          exit 1
        fi
      ;;
      "-i" | "--interface")
        c=$(( $c + 1 ))
        default_interface=${!c}
        ifconfig $default_interface 2>&1 >> /dev/null
        ifconfig_result=$?
        if [ $ifconfig_result -eq 1 ]; then
          echo Expected interface for -i parameter, found $default_interface.
          exit 1
        fi
      ;;
      "--ndisc")
        mode=ARP
      ;;
      *)
        echo Unknown parameter \"$parameter\"
        exit 1
      ;;
    esac
    c=$(( $c + 1 ))
  done
  hostname=${!c}
}

parse_parameters $@

# If there is something in brackets, assume it's an IPv6 address.
ipv6_address=$(echo $hostname | egrep '\[.+?\]')
if [ -n "$ipv6_address" ]; then
  ipmode=v6
  # If there's a colon, we've got a TCP request
  if [[ $hostname == *\]:* ]]; then
    portnum=$(echo $hostname | awk -F "]:" '{print $2}')
    hostname=$(echo $hostname | awk -F "]:" '{print $1}')
    hostname="${hostname}]"
    if [ "$mode" = "ARP" ]; then
      echo "Cannot use ARP mode with a port number."
      exit 2
    else
      mode=TCP
    fi
  fi
  ipv6_address=${hostname:1}
  ipv6_address=${ipv6_address%?}
else
  if [[ $hostname == *:* ]]; then
```

```
    portnum=$(echo $hostname | awk -F ":" '{print $2}')
    hostname=$(echo $hostname | awk -F ":" '{print $1}')
    if [ "$mode" = "ARP" ]; then
      echo "Cannot use ARP mode with a port number."
      exit 2
    else
      mode=TCP
    fi
  fi
fi

# If the hostname looks like an IPv4 address, do the right thing
if [ ! "$(echo $hostname | grep '[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}')" =
  ipmode=v4
  ipv4_address=$hostname
fi

# If this is true we should have an actual hostname
if [ $ipmode = "nodns" ]; then
ipv4_address=$(dig +search +short $hostname | grep '[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\
if [ -n "$ipv4_address" ]; then
  ipmode=v4
fi

ipv6_address=$(dig +search +short $hostname AAAA | grep \: | tail -n1)
if [ -n "$ipv6_address" ]; then
  ipmode=v6
fi

if [ $ipmode = "nodns" ] && [ "$hostname" = localhost ]; then
  ipv4_address="127.0.0.1"
  ipv6_address="::1"
  ipmode=v6
fi

fi

if [ $ipmode = nodns ]; then
  echo $programname: unknown host $hostname
  exit 2
fi

tcp_cmd_v6="nmap -6"
tcp_cmd_v4="nmap"
tcp_parameters="-PS -PN -p $portnum"

icmp_cmd_v6="ping6"
icmp_cmd_v4="ping"
icmp_parameters="-c1"

if [ "$default_interface" = "" ]; then
  # Get the default interface for IPv4 and use that.
  interface="$(ip route | grep default | awk '{print $5}')"
```

```
else
  interface="$default_interface"
fi
arp_cmd_v6="ndisc6"
arp_cmd_v6_parameters="$interface"
arp_cmd_v4="arping -c 1 -I $interface"

function create_command {
  full_mode="${mode}_${ipmode}"
  case $full_mode in
    TCP_v6*)
      ipaddress=$ipv6_address
      command="$tcp_cmd_v6 $tcp_parameters $ipaddress"
      parsecmd="tcp_parse"
    ;;
    TCP_v4*)
      ipaddress=$ipv4_address
      command="$tcp_cmd_v4 $tcp_parameters $ipaddress"
      parsecmd="tcp_parse"
    ;;
    ICMP_v6*)
      ipaddress=$ipv6_address
      command="$icmp_cmd_v6 $icmp_parameters $ipaddress"
      parsecmd="icmp_v6_parse"
      ipaddress=$ipv6_address
    ;;
    ICMP_v4*)
      ipaddress=$ipv4_address
      command="$icmp_cmd_v4 $icmp_parameters $ipaddress"
      parsecmd="icmp_v4_parse"
    ;;
    ARP_v6*)
      ipaddress=$ipv6_address
      command="$arp_cmd_v6 $ipaddress $arp_cmd_v6_parameters"
      parsecmd="arp_v6_parse"
    ;;
    ARP_v4*)
      ipaddress=$ipv4_address
      command="$arp_cmd_v4 $ipaddress"
      parsecmd="arp_v4_parse"
    ;;
  esac
}

function process_results {
  if [ "$commandresult" = 0 ]; then
    successful=$((successful+1))
#    if [ ! $mode = "ARP" ]; then
      latencytotal=$(echo $latencytotal + $latency | bc -l)
      if [ $(echo "$latency>$latencymax"|bc) = 1 ] || [ $successful -eq 1 ] ; then
        latencymax=$latency
      fi
      if [ $(echo "$latency<$latencymin"|bc) = 1 ] || [ $successful -eq 1 ]; then
```

```
        latencymin=$latency
      fi
#    fi
  fi
}

function tcp_parse {
  state=$(echo "$response" | grep ${portnum}/tcp | awk '{print $2}')
  if [ "$state" = "open" ]; then
    commandresult=0
    size="TCP Handshake"
    latency=$(echo "$response" | grep latency | awk '{print $4}')
    if [ "$latency" = "" ]; then
      latency=-1
    else
      latency=$(echo ${latency:1})
      latency=$(echo ${latency%?})
      latency=$(echo "scale=1; $latency * 1000 / 1" | bc -l)
    fi
    latency_units=ms
  else
    commandresult="($state)"
  fi
  process_results
}

function icmp_v6_parse {
  case "$commandresult" in
    1)
      commandresult="(No response)"
    ;;
    2)
      commandresult="(Network or host unreachable)"
    ;;
  esac
  if [ "$(echo "$response" | grep unreachable)" = "" ]; then
    size=$(echo $response | awk '{print $6 " " $7}')
    latency=$(echo $response | awk '{print $12}')
    latency_units=$(echo $response | awk '{print $13}')
    latency=$(echo $latency | awk -F "=" '{print $2}')
  else
    commandresult="(Unreachable)"
  fi
  process_results
}

function icmp_v4_parse {
  case "$commandresult" in
    1)
      commandresult="(No response)"
    ;;
    2)
      commandresult="(Network or host unreachable)"
```

```
    ;;
  esac
  if [ "$(echo "$response" | grep Unreachable)" = "" ]; then
    size=$(echo $response | awk '{print $8 " " $9}')
    latency=$(echo $response | awk '{print $14}')
    latency_units=$(echo $response | awk '{print $15}')
    latency=$(echo $latency | awk -F "=" '{print $2}')
  else
    commandresult="(Unreachable)"
  fi
  process_results
}

function arp_v6_parse {
  if [ $commandresult = 0 ]; then
    latency=$(echo "scale=3; (($stop_time - $start_time) * 1000 / 1)" | bc)
    latency_units=ms
    echo $latency
    mac_address=$(echo "$response" | grep Target | awk '{print $4}')
  else
    mac_address="timed out"
  fi
  process_results
}

function arp_v4_parse {
  if [ $commandresult = 0 ]; then
    latency=$(echo "$response" | grep -w reply | awk '{print $NF}' | sed 's/ms//g')
    latency_units=ms
    echo $latency
    mac_address=$(echo $(echo "$response" | awk -F\[ '{print $2}' | awk -F\] '{print $1}'))
  else
    mac_address="timed out"
  fi
  process_results
}

function send_ping {
  sequence=0
  successful=0
  latencymin=0
  latencymax=0
  latencytotal=0
  if [ $mode = TCP ]; then
    echo "PING $hostname ($ipaddress) via $mode on port $portnum using IP${ipmode} on $inter
  else
    echo "PING $hostname ($ipaddress) via $mode using IP${ipmode} on $interface."
  fi
  while [ $count -eq -1 ] || [ $count -gt $sequence ]
  do
    start_time=$(date +%s.%N)
    response=$($command 2> /dev/null)
    stop_time=$(date +%s.%N)
```

```
    commandresult=$?
    $parsecmd
    sequence=$((sequence+1))
    if [ $mode = "ARP" ]; then
        echo ARP from $hostname \($ipaddress\): seq=$sequence time=$latency $latency_units m
    else
      if [ "$commandresult" = 0 ]; then
        echo $size from $hostname \($ipaddress\): seq=$sequence time=$latency $latency_units
      else
        echo From $hostname \($ipaddress\): seq=$sequence $commandresult
      fi
    fi
    sleep 1
  done
  exit_function
}

function exit_function {
  endtime=$(date +%s.%N)
  timediff=$(echo "($endtime - $starttime)*1000/1" | bc)
  if [ $successful -eq 0 ]; then
    latencyavg=0
    percentloss=100
  else
    latencyavg=$(echo "scale=1; $latencytotal / $successful" | bc -l)
    percentloss=$(echo "scale=0; ($sequence - $successful) / $successful * 100" | bc -l)
  fi
  echo ""
  echo --- $hostname ping statistics ---
  echo $sequence connections attempted, $successful successful, ${percentloss}\% loss, time
  echo rtt min/avg/max/mdev = $latencymin/$latencyavg/$latencymax/$latencymdev ms
  if [ 0 -eq 0 ] && [ $successful -eq 0 ] && [ $ipmode = "v6" ] && [ -n "$ipv4_address" ]; t
    echo ""
    echo "-------------- Attempts to connect on IPv6 failed, trying IPv4 --------------"
    echo ""
    if [ $mode = TCP ]; then
      $0 ${ipv4_address}:${portnum}
    else
      $0 $ipv4_address
    fi
  fi
  exit
echo $command
}

trap "exit_function" SIGINT SIGTERM
create_command
starttime=$(date +%s.%N)
send_ping
```

Running the command without any parameters brings up a help window with some examples. The script takes a single parameter, the hostname and optional port number of the host you'd like to check on. It uses pretty standard notation, *hostname[:portnum]*. Some examples:

```
$ uping www.google.com
PING www.google.com (173.194.33.104) via ICMP using IPv4.
64 bytes from www.google.com (173.194.33.104): seq=1 time=27.6 ms
64 bytes from www.google.com (173.194.33.104): seq=2 time=30.3 ms
64 bytes from www.google.com (173.194.33.104): seq=3 time=27.8 ms
^C
--- www.google.com ping statistics ---
3 connections attempted, 3 successful, 0% loss, time 3130 ms
rtt min/avg/max/mdev = 27.6/28.5/30.3/ ms
```

In the above example we're using ICMP to talk to Google on IPv4. The hostname doesn't resolve to an IPv6 address and we did not specify a port number.

```
$ uping www.google.com:80
PING www.google.com (173.194.33.104) via TCP on port 80 using IPv4.
TCP Handshake from www.google.com (173.194.33.104): seq=1 time=37.0 ms
TCP Handshake from www.google.com (173.194.33.104): seq=2 time=40.0 ms
TCP Handshake from www.google.com (173.194.33.104): seq=3 time=37.0 ms
^C
--- www.google.com ping statistics ---
3 connections attempted, 3 successful, 0% loss, time 2766 ms
rtt min/avg/max/mdev = 37.0/38.0/40.0/ ms
```

We're talking to Google again, but now we've specified port 80.

```
$ uping ipv6.google.com:80
PING ipv6.google.com (2001:4860:800f::68) via TCP on port 80 using IPv6.
TCP Handshake from ipv6.google.com (2001:4860:800f::68): seq=1 time=44.0 ms
TCP Handshake from ipv6.google.com (2001:4860:800f::68): seq=2 time=40.0 ms
TCP Handshake from ipv6.google.com (2001:4860:800f::68): seq=3 time=36.0 ms
^C
--- ipv6.google.com ping statistics ---
3 connections attempted, 3 successful, 0% loss, time 2641 ms
rtt min/avg/max/mdev = 36.0/40.0/44.0/ ms
```

Now we're pinging port 80 on Google's IPv6 server.

```
$ uping 72.163.4.161:443
PING 72.163.4.161 (72.163.4.161) via TCP on port 443 using IPv4.
TCP Handshake from 72.163.4.161 (72.163.4.161): seq=1 time=84.0 ms
TCP Handshake from 72.163.4.161 (72.163.4.161): seq=2 time=85.0 ms
TCP Handshake from 72.163.4.161 (72.163.4.161): seq=3 time=85.0 ms
^C
--- 72.163.4.161 ping statistics ---
3 connections attempted, 3 successful, 0% loss, time 3006 ms
rtt min/avg/max/mdev = 84.0/84.6/85.0/ ms
```

Now we're pinging port 443 on a specific IP address.

```
$ uping [2620:0:ef0:13::20]:80
PING [2620:0:ef0:13::20] (2620:0:ef0:13::20) via TCP on port 80 using IPv6.
TCP Handshake from [2620:0:ef0:13::20] (2620:0:ef0:13::20): seq=1 time=110.0 ms
TCP Handshake from [2620:0:ef0:13::20] (2620:0:ef0:13::20): seq=2 time=110.0 ms
TCP Handshake from [2620:0:ef0:13::20] (2620:0:ef0:13::20): seq=3 time=110.0 ms
^C
--- [2620:0:ef0:13::20] ping statistics ---
3 connections attempted, 3 successful, 0% loss, time 3084 ms
rtt min/avg/max/mdev = 110.0/110.0/110.0/ ms
```

Finally, when specifying an IPv6 address (in this case, NetFlix's address) you'll need to enclose it in brackets. This is often only required when a port is specified, but because I'm way too lazy to come up with a regex for IPv6 addresses you're stuck with it anytime you want to use IPv6 without DNS. Note to anyone involved, use DNS (../freedns) with IPv6.

4.2.1. ARP/NDISC Pinging

The **uping** command now also supports ARP/Neighbor Discovery (IPv6 ARP) pinging. This obviously only works if the host is on the local subnet and can provide a quick interface to grabbing a remote host's MAC address. It uses the commands **arping** or **ndisc6** to perform this function.

Syntax is designed to be pretty easy. It will try to guess the appropriate interface using your IPv4 routing table, otherwise you can specify it using a parameter like `-i eth0`. To do an ARP ping on either IPv4 or IPv6 you can specify **uping --arp -i eth1 mylaptop.mydomain.com**.

4.2.2. MAC Scanner

I've tossed in an add-on utility that doesn't belong here because it's not universal, but is also dumb enough not to deserve its own package name. I call it **mac_scan** and the only argument it takes is an interface (it will default to whichever interface owns your default gateway). Essentially all it does is practice bitwise math in BASH (yes, MUCH easier to do in Perl but I'd rather not add the Perl dependency now). It will calculate your network and subnet, then ping every IP address between those two (non-inclusive). It then waits for 5 seconds and displays the ARP table. No broadcast ping because many devices/OSes don't respond to them however I do send all 254 pings out at about the same time. Note that I do check and I won't scan anything larger than a class C, edit the code if you think you want to do that. Skip past listing.

Example 3. mac_scan

```
#!/bin/bash

if [ $# -eq 1 ]; then
  interface="$1"
else
```

```
  interface="$(ip route get 1.1.1.1 | awk '{print $5}')"
fi
interface_info=$(ifconfig $interface)

function ip2int {
  echo $1 | awk -F\. '{printf "%1.0f", 16777216 * $1 + 65536* $2 + 256 * $3 + $4 }'
}

function int2ip {
  ip_int=$1
  first_octet=$(( $ip_int / 16777216 ))
  ip_int=$(( $ip_int - 16777216 * $first_octet ))
  second_octet=$(( $ip_int / 65536 ))
  ip_int=$(( $ip_int - 65536 * $second_octet ))
  third_octet=$(( $ip_int / 256 ))
  fourth_octet=$(( $ip_int - 256 * $third_octet))
  echo -n $first_octet.$second_octet.$third_octet.$fourth_octet
}

inet_address=$(echo "$interface_info" | grep "inet addr" | awk -F\: '{print $2}' | awk '{pri
subnet_mask=$(echo "$interface_info" | grep "inet addr" | awk -F\: '{print $4}' | awk '{prin

inet_int=$(ip2int $inet_address)
subnet_int=$(ip2int $subnet_mask)
network_int=$(( $inet_int & $subnet_int ))
inv_subnet_int=$(( $subnet_int ^ 4294967295 ))
bcast_int=$(( $inet_int | $inv_subnet_int ))

echo "Scanning network, this will take a few seconds:
iface $interface, inet $inet_address, subnet $subnet_mask
"

if [ $(( $bcast_int - $network_int )) -le 256 ]; then
  c=$(( $network_int + 1 ))
  while [ $c -lt $bcast_int ]; do
    ping -c1 $(int2ip $c) 2>&1 >> /dev/null &
    c=$(( $c + 1 ))
  done
  sleep 5
  arp -n | grep -v incomplete
else
  echo Sorry, your network has too many hosts
  exit 2
fi
```

Why don't I like this program? Well, ARP is a pretty messy way of trying to find out all the IP addresses on a network. What's even worse is that on a typical /64 IPv6 network at a billion packets a second (that's assuming you've got at least terabit Ethernet running) it would still take 584 years to figure out all the addresses on a subnet. Now **ping6** has a neat option to poll all of the link addresses by running **ping6 -I eth0 ff02::1**, but remember that this really just gives you a list of all the MAC addresses rather than the useful service addresses.

However, I replaced my router recently which means that I lost my old MAC to IP bindings. Running this program can help me find a lost device so I know what needs to be changed.

5. TCP Traceroute

Luckily, there is a Ubuntu package for **tcptraceroute** (and its hotter sister, **tcptraceroute6**) which is both easy to use and does exactly what I want. To install it, simply type **sudo apt-get install tcptraceroute ndisc6** and then run **tcptraceroute[6] <hostname> <port_number>** to execute it.

How it works is by sending three TCP SYN packets to the destination with a TTL (Time To Live) of 1. When the packet hits the router it decrements the TTL by one and, once the TTL reaches 0, an ICMP packet is sent back to indicate that it failed to reach the destination (so you still need ICMP to be able to come back to you). The source address of this ICMP packet gives away the IP address of the intermediate router. The **tcptraceroute** program then repeats this process, incrementing the TTL by one for each hop which lets it go one step further until the SYN is responded to with a SYN/ACK which makes **tcptraceroute** send a RST to terminate the connection. Because it uses TCP in the forward direction instead of UDP like regular traceroute does you should be able to be more successful.

In order to simplify the selection of the various traceroute options, I've created a slightly simpler script based on the syntax and structure of **uping2**, but I let the existing utilities do all the work for me. I've called it **utrace2** which I link to **utrace**. To run it, use the same syntax as **uping** with a command like **utrace <hostname>[:<port_number>]**.

The same limitations as above existing for **utrace**, the individual tools are much more powerful and flexible since I don't map the command-line parameters. Additionally, **utrace** isn't as consistent in the presented data since I simply call the external programs directly. The program listing below may shed some light on what I'm doing. Skip past listing.

Example 4. utrace2

```
#!/bin/bash

programname=‘basename $0‘

if [ $# -eq 0 ]; then
  echo Usage\:
  echo "  $programname hostname[:portnum]"
  echo ""
  echo "hostname can be an actual hostname, an IP address, or an IPv6 address"
  echo "Examples:"
  echo "$programname www.ebower.com           : sends UDP trace to www.ebower.com"
  echo "$programname www.ebower.com:443       : sends TCP trace on port 443"
  echo "$programname 192.168.1.1              : sends UDP trace to 192.168.1.1"
  echo "$programname [2001:4830:113c::80]     : sends UDP trace to 2001:4830:113c::80"
```

```
  echo "$programname [2001:4830:113c::80]:80  : sends TCP trace on port 80"
  exit 0
fi

hostname=$1

mode=UDP
ipmode=nodns

ipv6_address=$(echo $hostname | egrep '\[.+?\]')
if [ -n "$ipv6_address" ]; then
  ipmode=v6
  if [[ $hostname == *\]:* ]]; then
    portnum=$(echo $hostname | awk -F "]:" '{print $2}')
    hostname=$(echo $hostname | awk -F "]:" '{print $1}')
    hostname="${hostname}]"
    mode=TCP
  fi
  ipv6_address=${hostname:1}
  ipv6_address=${ipv6_address%?}
else
  if [[ $hostname == *:* ]]; then
    portnum=$(echo $hostname | awk -F ":" '{print $2}')
    hostname=$(echo $hostname | awk -F ":" '{print $1}')
    mode=TCP
  fi
fi

# If the hostname looks like an IPv4 address, do the right thing
if [ ! "$(echo $hostname | grep '[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}')" =
  ipmode=v4
  ipv4_address=$hostname
fi

# If this is true we should have an actual hostname
if [ $ipmode = "nodns" ]; then
ipv4_address=$(dig +search +short $hostname | grep '[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\
if [ -n "$ipv4_address" ]; then
  ipmode=v4
fi

ipv6_address=$(dig +search +short $hostname AAAA | grep \: | tail -n1)
if [ -n "$ipv6_address" ]; then
  ipmode=v6
fi

fi

if [ $ipmode = nodns ]; then
  echo $programname: unknown host $hostname
  exit 2
fi
```

```
tcp_cmd_v6="tcptraceroute6"
tcp_cmd_v4="tcptraceroute"
# Reserved for future use
tcp_parameters=""

udp_cmd_v6="tracepath6"
udp_cmd_v4="tracepath"
tracepath_version=$( tracepath 2>&1 )
if [ "$(echo $tracepath_version | grep '\[\-b\]')" = "" ]; then
  upd_parameters=""
else
  udp_parameters="-b"
fi

function create_command {
  full_mode="${mode}_${ipmode}"
  case $full_mode in
    TCP_v6*)
      ipaddress=$ipv6_address
      command="$tcp_cmd_v6 $tcp_parameters $ipaddress $portnum"
      parsecmd="tcp_parse"
    ;;
    TCP_v4*)
      ipaddress=$ipv4_address
      command="$tcp_cmd_v4 $tcp_parameters $ipaddress $portnum"
      parsecmd="tcp_parse"
    ;;
    UDP_v6*)
      ipaddress=$ipv6_address
      command="$udp_cmd_v6 $udp_parameters $ipaddress"
      parsecmd="udp_v6_parse"
    ;;
    UDP_v4*)
      ipaddress=$ipv4_address
      command="$udp_cmd_v4 $udp_parameters $ipaddress"
      parsecmd="udp_v4_parse"
    ;;
  esac
}

create_command
if [ $mode = TCP ]; then
  echo Initiating trace to $hostname \($ipaddress\), port $portnum, using $mode over IP$ipmo
else
  echo Initiating trace to $hostname \($ipaddress\) using $mode over IP$ipmode
fi

$command |grep -v traceroute
```

A. Installation

Most Ubuntu users should strongly consider using my Launchpad PPA which will help keep your package up-to-date. You can install it with the following commands:

```
sudo add-apt-repository ppa:ubuntu-ebower/ebower
sudo apt-get update
sudo apt-get install uping
```

If you're not fortunate enough to be able to use this, you can download the packages here:

Table A-1. Download links for "uping"

Distro Type	i386	amd64
Debian	uping_0.2.7.1_i386.deb	uping_0.2.7.1_amd64.deb
RedHat	uping-0.2.7.1-2.i386.rpm	uping-0.2.7.1-2.x86_64.rpm
Other	uping_0.2.7.1.tar.gz	uping_0.2.7.1.tar.gz

B. About Me

My name is Jeff Bower, I'm a technology professional (http://www.linkedin.com/in/jdbower) with more years of experience in the telecommunications industry than I'd care to admit. I tend to post with the username jdbower on various forums, including Komodo Kamado (http://komodokamado.com/forum/), Android Central (http://forum.androidcentral.com/), VirtualBox (http://forums.virtualbox.org/), and MakeMKV (http://www.makemkv.com/forum2/). Writing these documents is a hobby of mine, I hope you find them useful and feel free to browse more at https://www.ebower.com/docs.

I also enjoy cooking, especially outdoors with my Komodo Kamado (http://www.komodokamado.com) and using my Stoker (https://www.rocksbarbque.com). Take a look at my recipes stored at https://www.ebower.com/recipes.

If you've got any questions or feedback please feel free to email me at docs@ebower.com (mailto:docs@ebower.com) or follow me on Google+ (https://profiles.google.com/100268310848930740059) or Twitter (http://twitter.com/jdbower).