Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.829 Fall 2002 **Networking Tools** September 13, 2001

Overview. In this tutorial, we will cover tools that are commonly used in networking research (not to mention in the "real world" for debugging). We begin with `netstat` and `tcpdump`, describing briefly how to use each of these tools by looking at a live TCP connection and making sense of their output. We'll use netstat to describe one of the subtleties of the TCP state machine.

Next, we'll look at a couple of lookup-based tools, `dig` and `whois`, that provide information about DNS information about hosts, and registry information about names and networks, respectively.

We will then move on to standard network measurement tools, `ping` and, more interestingly, `traceroute`, describing how they work and what they do (and don't) tell you.

Finally, for those of you who don't know perl, I'll show how to "tie things together" by demonstrating a simple perl script that takes our trace and counts bytes to destination IP addresses and ASes.

1 `tcpdump` and `netstat`

How does tcpdump work? For BSD-derived kernels, the Berkeley Packet Filter (BPF) is available. The filter places the device driver into promiscuous mode, receives from the driver receive all transmitted and received packets. These packets are then run through a user-specified filter, so that only the packets that the user specifies as interesting will be passed to the user process. What about the costliness to constantly trapping to the kernel for small reads? A timeout value with batch reads solves this problem.

Filtering is done in the kernel. This limits the amount of data that must be copied from the kernel into user space.

Here is what the output from `tcpdump host -i eth0 aros.ron.lcs.mit.edu port ssh` from my machine will look like:

```
08:52:34.173548 ginseng.lcs.mit.edu.3524 > aros.ron.lcs.mit.edu.ssh: S
821658836:821658836(0) win 32120 <mss 1460,sackOK,timestamp
203984870[|tcp]> (DF)
08:52:34.256660 aros.ron.lcs.mit.edu.ssh > ginseng.lcs.mit.edu.3524: S
2025096886:2025096886(0) ack 821658837 win 57344 <mss 1460,nop,wscale
0,nop,nop,timestamp[|tcp]>
08:52:34.256691 ginseng.lcs.mit.edu.3524 > aros.ron.lcs.mit.edu.ssh:
. ack 1 win 32120 <nop,nop,timestamp 203984878 3657001720> (DF)
08:52:34.342643 aros.ron.lcs.mit.edu.ssh > ginseng.lcs.mit.edu.3524: P
1:41(40) ack 1 win 57920 <nop,nop,timestamp 3657001806 203984878> (DF)
08:52:34.342671 ginseng.lcs.mit.edu.3524 > aros.ron.lcs.mit.edu.ssh:
. ack 41 win 32080 <nop,nop,timestamp 203984887 3657001806> (DF)
```

```
<CUT>

08:53:22.367079 ginseng.lcs.mit.edu.3524 > aros.ron.lcs.mit.edu.ssh: F
715:715(0) ack 1381 win 32120 <nop,nop,timestamp 203989689 3657049834>
(DF) [tos 0x10]
08:53:22.449727 aros.ron.lcs.mit.edu.ssh > ginseng.lcs.mit.edu.3524:
. ack 716 win 57908 <nop,nop,timestamp 3657049917 203989689> (DF) [tos
0x10]
08:53:22.451677 aros.ron.lcs.mit.edu.ssh > ginseng.lcs.mit.edu.3524: F
1381:1381(0) ack 716 win 57908 <nop,nop,timestamp 3657049919 203989689>
(DF) [tos 0x10]
08:53:22.451701 ginseng.lcs.mit.edu.3524 > aros.ron.lcs.mit.edu.ssh:
. ack 1382 win 32120 <nop,nop,timestamp 203989698 3657049919> (DF) [tos
0x10]
```

What's going on with the timestamp option? For example, why not just mark a packet with the current time it was sent? (Requires clock synchronization.)

Information included in each line: a timestamp, source and destination (IP address and port), TCP flags, segment info (start, end, size), acks, window size, various other TCP options. The "P" flag stands for "push" the received data to the application as soon as possible. Notice the 3-way handshake. Also, notice the process of tearing down a connection:

```
tcp      0      0 ginseng.lcs.mit.ed:3524 aros.ron.lcs.mit.ed:ssh ESTABLISHED
tcp      0      0 ginseng.lcs.mit.ed:3524 aros.ron.lcs.mit.ed:ssh TIME_WAIT
```

Briefly describe the state transition diagram as the summary of the "rule set" for how TCP behaves.

Why does the connection go into the time wait state? Note this is also called the "2MSL" (maximum segment lifetime) state. The end that performs the active close does not know if the final ACK was ever received, so it doesn't know that the other side closed the connection. Just because the FIN arrives, we can't assume that the connection is closed — segments may arrive out of order.

Can you think of various attacks on TCP? Sequence number hijacking is one example. SYN flood attacks is another? How would you defend against a SYN flood? One approach is something called "SYN Cookies", whereby the sender does not set up state for a connection until after the three-way handshake is completed (rather than going into the SYN received state immediately after receiving the first SYN). This will be described in lecture in the coming weeks.

2 Lookup Tools: `dig` and `whois`

`dig` is a DNS lookup utility. It's similar to `nslookup`, but dig is more powerful. You can query for various types of records. Most common (and default) is the "A" record, which maps a DNS name to an IP address.

Here is some example output for a lookup on my machine. Notice that the query is for an "A" record. `dig` also returns a bunch of other useful information. For example, it returns the "NS" records for `lcs.mit.edu.` The NS records return the authoritative DNS servers for a particular domain. Notice I also get the "A" records for all of those machines. Why is there a dot at the end of every domain? Because this tells you the name is fully qualified (try looking up `ai` vs. `ai.` on MIT machines).

The numbers such as "1800" describe how long a local DNS server should cache the result of that answer.

Explain what the output looks like when the name doesn't resolve to anything. Explain the same for reverse DNS, partially qualified domain names, etc.

```
9:38am:ginseng:~% dig ginseng.lcs.mit.edu

; <<>> DiG 9.2.1 <<>> ginseng.lcs.mit.edu
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 21855
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 4, ADDITIONAL: 9

;; QUESTION SECTION:
;ginseng.lcs.mit.edu.           IN      A

;; ANSWER SECTION:
ginseng.lcs.mit.edu.    1800    IN      A       18.31.0.38

;; AUTHORITY SECTION:
lcs.mit.edu.            1800    IN      NS      mintaka.lcs.mit.edu.
lcs.mit.edu.            1800    IN      NS      fedex.ai.mit.edu.
lcs.mit.edu.            1800    IN      NS      ossipee.lcs.mit.edu.
lcs.mit.edu.            1800    IN      NS      lampang.lcs.mit.edu.

;; ADDITIONAL SECTION:
mintaka.lcs.mit.edu.    1800    IN      A       18.26.0.36
fedex.ai.mit.edu.       12769   IN      A       192.148.252.43
ossipee.lcs.mit.edu.    1800    IN      A       18.26.0.18
ossipee.lcs.mit.edu.    1800    IN      A       18.24.10.7
ossipee.lcs.mit.edu.    1800    IN      A       18.111.0.2
lampang.lcs.mit.edu.    1800    IN      A       18.24.0.120
lampang.lcs.mit.edu.    1800    IN      A       18.24.1.120
lampang.lcs.mit.edu.    1800    IN      A       18.24.2.5
lampang.lcs.mit.edu.    1800    IN      A       18.24.4.120

;; Query time: 3 msec
;; SERVER: 18.26.0.18#53(18.26.0.18)
;; WHEN: Thu Sep 12 09:38:56 2002
;; MSG SIZE  rcvd: 286
```

Other things you might see are types like CNAME. Often people have aliases, or multiple names for one machine. This is implemented using canonical names, which are expressed in DNS via a

CNAME record. This is shown below:

```
;; ANSWER SECTION:
bgp.lcs.mit.edu.        1800    IN      CNAME   mit-network-monitor.lcs.mit.edu.
mit-network-monitor.lcs.mit.edu. 1800 IN A      18.31.0.51
```

One can also use `dig` to look up the mail server for a particular domain. Let's say someone wanted to send mail to `feamster@lcs.mit.edu`; how does their mail program find the mail server responsible for delivering mail to `lcs.mit.edu`? This is expressed via an MX record. I can query for this record, for example, by typing `dig lcs.mit.edu MX`. What's up with the numbers? They express preference for one mail server over the other.

```
;; ANSWER SECTION:
lcs.mit.edu.            1800    IN      MX      10 fedex.ai.mit.edu.
lcs.mit.edu.            1800    IN      MX      1 mintaka.lcs.mit.edu.
```

You can also use `dig` to do reverse lookups. `dig -x` is the easiest way to do this. Dig can also be used in batch mode. See the man page for more details.

`whois` is also a useful tool that allows you to query any whois server. There are a few types of whois servers. One is the "nic" servers, that typically tell you information about who a particular name is registered to. The default whois server on my machine is `whois.crsnic.net`. This whois server then issues a query to a whois server that is responsible for registering that name. An excerpt of the output is below:

```
9:46am:ginseng:~% whois mit.edu
[whois.crsnic.net]

   Domain Name: MIT.EDU
   Registrar: EDUCAUSE
   Whois Server: whois.educause.net
   Referral URL: http://www.educause.edu/edudomain
   Name Server: BITSY.MIT.EDU
   Name Server: STRAWB.MIT.EDU
   Name Server: W20NS.MIT.EDU
   Updated Date: 24-may-2002

[whois.educause.net]
Domain Name: MIT.EDU

Registrant:
   Massachusetts Institute of Technology
   Cambridge, MA 02139
   UNITED STATES
```

Whois can also be used to query the address registries, ARIN (North America), APNIC (Asia), and RIPE (Europe). For example, let's say I wanted to know more about the address `18.31.0.38`:

```
9:59am:ginseng:% whois -h whois.arin.net 18.31.0.38
[whois.arin.net]

OrgName:    Massachusetts Institute of Technology
OrgID:      MIT-2

NetRange:   18.0.0.0 - 18.255.255.255
CIDR:       18.0.0.0/8
NetName:    MIT
NetHandle:  NET-18-0-0-0-1
Parent:
NetType:    Direct Assignment
NameServer: STRAWB.MIT.EDU
NameServer: W20NS.MIT.EDU
NameServer: BITSY.MIT.EDU
Comment:
RegDate:
Updated:    1998-09-26

TechHandle: JIS-ARIN
TechName:   Schiller, Jeffrey
TechPhone:  +1-617-253-8400
TechEmail:  jis@mit.edu

# ARIN Whois database, last updated 2002-09-11 19:05
# Enter ? for additional hints on searching ARIN's Whois database.
```

Note also the date at which the database was last updated, and when the record was last updated. This can give you some idea about the accuracy of the information.

Besides whois servers that are used to resolve names into administrative domains, there exist whois servers that help map numbers into administrative domains. Specifically, the "numbers" (i.e., IP addresses and AS numbers) are managed by the following organizations:

- RIPE: Reseaux IP Europeans
- ARIN: American Registry for Internet Numbers (also does Africa)
- APNIC: Asia Pacific Network Information Center (and Australia)

If I wanted to know something about some IP address or AS number at MIT, I could then as the ARIN whois server (whois.arin.net). The ARIN whois server has a lot of nice options whereby you can limit the query to a specific record type. Let's say I wanted to look up "AS 701":

```
10:07am:ginseng:~% whois -h whois.arin.net "a 7018"
[whois.arin.net]

OrgName:  AT&T WorldNet Services
```

```
OrgID:    ATTW

ASNumber: 7018
ASName:   ATT-INTERNET4
ASHandle: AS7018
Comment:
RegDate:  1996-07-30
Updated:  2000-02-01
```

Queries on supernets are also supported; APINIC (`whois.apnic.net`) and RIPE (`whois.ripe.net`) work in a similar fashion. How should we go about finding the autonomous system for a particular network or IP address? There's another whois database, called RADB, that keeps this information relatively up-to-date. Note that this is how `traceroute -A` resolves IP addresses to AS's.

```
10:06am:ginseng:~% whois -h whois.radb.net 18.31.0.38
[whois.radb.net]
route:         18.0.0.0/8
descr:         Massachusetts Institute of Technology
               1 Amherst Street
               Cambridge
               MA 02139, USA
origin:        AS3
member-of:     RS-COMM_NSFNET
mnt-by:        MAINT-AS3
changed:       nsfnet-admin@merit.edu 19950118
source:        RADB
```

3 Measurement Tools: `ping` and `traceroute`

`ping` is a simple program that sends an ICMP "Echo Request" message to a machine and waits to hear an ICMP "Echo Reply". It's basically useful for telling if a particular host is responding (and therefore that it's actually turned on, on the network, etc.). Ping also shows an RTT value for each request/reply pair, and a sequence number. Ping will give a summary of the loss rate as well. Note that the RTT may not be completely accurate, since ICMP is not on the fast path for many routers. Interesting options include "f" (flood, sometimes requires root), "c" (send a certain number of packets), "s" (specify packet size, sometimes useful for looking at fragmentation).

Try sending pings of large packet size, and watching the IP ID in tcpdump.

`traceroute` is a tool for getting some idea about the route that a particular host sees to a certain destination. Traceroute sends an IP packet with increasing TTL values (actually, 3 per TTL value), and listens for ICMP "TIME_EXCEEDED" responses for each hop along the path to the host. Some important things to note about traceroute:

- Each hop specified an interface (from which the time-exceeded message was sent), *not* a host or a router. A path may traverse more than one hop on a particular router.

- It may be the case that the source of the time-exceeded message is set to the IP address of the *outgoing* interface on the return path to the host, rather than the interface that the packet arrived on. This can produce some surprising results.
- A failure in a traceroute does not indicate a failure at that point (necessarily). It could indicate a failure along the reverse path, for example. Additionally, "*" does not indicate a failure condition, necessarily; this might mean, for example, that the router does not respond to ICMP messages.

Note that the version of traceroute on most machines doesn't support the "-A" option. You can either do this manually (as above), fine the RPM or appropriate existing package for your machine, or download the "Nanog Traceroute" from `http://nms.lcs.mit.edu/6.829/other/traceroute-nanog.tar.gz`, and compile it for your particular machine (note that traceroute either requires superuser privileges to run, or you must set it to "setuid root", or you will get permission errors).

```
10:20am:ginseng:~% traceroute -A aros.ron.lcs.mit.edu
traceroute to aros.ron.lcs.mit.edu (206.197.119.141), 30 hops max, 40 byte packets
 1  anacreon (18.31.0.1) [AS3]  1 ms  1 ms  1 ms
 2  radole (18.24.10.3) [AS3]  2 ms  1 ms  1 ms
 3  B24-RTR-1-LCS-LINK.MIT.EDU (18.201.1.1) [AS3]  7 ms  2 ms  2 ms
 4  EXTERNAL-RTR-2-BACKBONE.MIT.EDU (18.168.0.27) [AS3]  2 ms  2 ms  2 ms
 5  p4-0.bstnma1-cr5.bbnplanet.net (4.24.88.49) [AS1]  2 ms  2 ms  2 ms
 6  so-4-3-0.bstnma1-nbr2.bbnplanet.net (4.24.4.202) [AS1]  2 ms  3 ms  1 ms
 7  p9-0.nycmny1-nbr2.bbnplanet.net (4.24.6.50) [AS1]  8 ms  9 ms  8 ms
 8  p1-0.nycmny1-cr10.bbnplanet.net (4.24.8.170) [AS1]  8 ms  9 ms  9 ms
 9  a1-0.xnycmny4-uunet.bbnplanet.net (4.0.6.142) [AS1]  10 ms  9 ms  8 ms
10  0.so-6-0-0.XL1.NYC9.ALTER.NET (152.63.18.226) [AS701]  9 ms  8 ms  10 ms
11  0.so-4-0-0.TL1.NYC9.ALTER.NET (152.63.0.173) [AS701]  9 ms  9 ms  8 ms
12  0.so-1-0-0.TL1.SLT4.ALTER.NET (152.63.1.157) [AS701]  74 ms  75 ms  75 ms
13  0.so-3-0-0.XL1.SLT4.ALTER.NET (152.63.9.69) [AS701]  74 ms  76 ms  74 ms
14  187.ATM8-0-0.GW1.SLT1.ALTER.NET (152.63.91.85) [AS701]  75 ms  76 ms  76 ms
15  aros-gw.customer.ALTER.NET (157.130.107.34) [AS701]  85 ms  83 ms  83 ms
16  zeus.aros.net (207.173.16.111) [AS6521/AS5650]  84 ms  84 ms  86 ms
17  aros.ron.lcs.mit.edu (206.197.119.141) [AS6521/AS5650]  88 ms  83 ms  84 ms
```

4 Perl

In this section, we'll see how to use Perl (a popular scripting language) to perform two analysis tasks on a tcpdump file:

- Count the number of bytes transferred from one host to another
- Plot an ack trace

Note that the following code does not show you how to use some basic things, such as arrays and hashes, which will also be helpful for the problem set. For this information (and for more information about regular expressions, Perl, etc., a good book is *Programming Perl*, the O'Reilly book by Larry Wall. The Perl man pages (type "man perl" at a Unix prompt) are also an excellent reference.

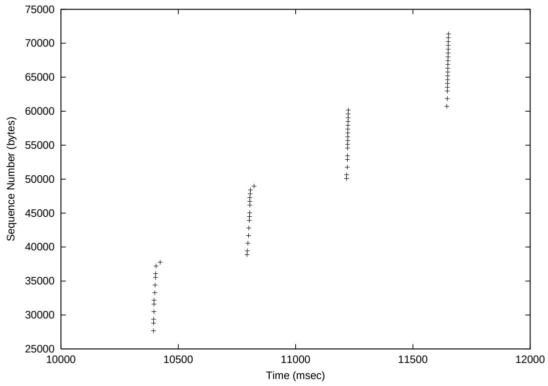
The following program does two things: it has a function to count bytes in a tcpdump trace, given the tcpdump trace file (in the appropriate trace dump format), and another function to produce an "ACK trace" for that tcpdump trace. The two functions are separated below for pedagogical purposes. I've shown this here to give you an idea of what perl looks like and how to do basic things that are commonly useful in networking research/analysis; this is not a tutorial on the language.

```
#!/usr/bin/perl

use strict;

# variables for program definitions

my $gnuplot = "/usr/local/bin/gnuplot";
my $tcpdump = "/usr/local/sbin/tcpdump";


# where my tracefile is, and the specified source and sink
# you might want to actually specify these types of things
# with an option, e.g., using the Getopt::Long package

my $trace = "/home/feamster/rv.tr";
my $source = "route-views.oregon-ix.net.telnet";
my $sink = "ginseng.lcs.mit.edu.3967";


sub calculate_bytes {

    # calculates the number of bytes transferred from the source to the
    # sink, given a tcpdump trace file

    my $total_bytes = 0;

    # open a file handle to read process output.
    open (F, "$tcpdump -r $trace |") || die "Can't open $trace: $!\n";
    while (<F>) {

# match on a specific pattern, based on the lines of tcpdump
# we want to look at traffic from the source to the sink in this case

if (/$source.*$sink.*\((\d+)\)/) {

    # add to the total number of bytes that which was matched by ()
    $total_bytes += $1;
}
```

```
    }
    printf "total bytes = %d\n", $total_bytes;
}


sub time_to_msec {

    # takes tcpdump-formatted time and outputs in milliseconds

    my ($time) = @_;

    # split into the hours,mins,secs part and the microseconds part, around "."
    my ($hms, $usec) = split(/\./, $time);

    # split the hours,mins,secs part into distinct values
    my ($hr,$min,$sec) = split(/:/,$hms);


    return( ((($hr*60+$min)*60 + $sec)*1000000+$usec)/1000);
}


sub plot_ack_trace {
    my ($starttime, $last_time);

    my $tmpfile = "/tmp/acktrace.dat";
    my $gpfile = "/tmp/acktrace.gp";
    my $pngfile = "/tmp/acktrace.png";


    # TMP is an output file handle where we will send the formatted data
    open (TMP, ">$tmpfile") || die "Can't open $tmpfile: $!\n";

    # open a file handle to read process output.
    open (F, "$tcpdump -r $trace |") || die "Can't open $trace: $!\n";
    while (<F>) {

# match on the timestamp, and the number followed by "ack"
if (/^(\d+:\d+:\d+\.\d+).*$sink.*$source.*ack (\d+)/) {
    my $timestamp = &time_to_msec($1);

    # normalize the time, so that the first timestamp is "0";
    $starttime = $timestamp if (!defined($starttime));

    my $window = $2;
    printf TMP "%lf %d\n", $timestamp-$starttime, $window;
    $last_time = $timestamp-$starttime;
}
```

```
    }
    close(F);
    close(TMP);


    # automated printing of gnuplot commands
    open (GPFILE, ">$gpfile") || die "Can't open $gpfile: $!\n";
    print GPFILE <<EOF_GP;
    set xlabel "Time (msec)";
    set ylabel "Sequence Number (bytes)";
    set term png color;
    set output "$pngfile";
    set xrange[10000:12000];
    plot"$tmpfile" using 1:2;
    set xrange 0:100;
EOF_GP
    close(GP);

    # system call to execute gnuplot and print our graph
    system("$gnuplot $gpfile");

}

&calculate_bytes();
&plot_ack_trace();

}

&calculate_bytes();
&ack_trace();
```

`calculate_bytes` prints some (hopefully correct) number of bytes sent from `routeviews` to `ginseng`. The `ack_trace` function produces the following figure (this is actually a zoom on the complete ACK trace for the whole connection):



Why does the trace look the way it does? It's basically because we're looking at the receiver's side: the sender fills the congestion window, we're ACK-ing a window's worth of packets, and the process repeats itself.