Chapter 7 – Ping

Packet InterNet Groper

Used to test to see if a host is reachable at all.
Historically, ping access was a necessary condition for any access
Now, it's not necessarily so....
- Some hosts don't acknowledge *pings*.
- Some firewalls filter *pings.*

Ping protocol is simply an ICMP exchange

ICMP type 8 code 0
- Ping request

ICMP type 0 code 0
- Ping reply

Here is an ping to California from about 1996

```
[L:\ACAD\CS481NET] ==> ping ftp.cdrom.com
PING wcarchive.cdrom.com: 56 data bytes
64 bytes from 192.216.191.11: icmp_seq=0. time=93. ms
64 bytes from 192.216.191.11: icmp_seq=1. time=94. ms
64 bytes from 192.216.191.11: icmp_seq=2. time=94. ms
```

This was done August 2001 (note similar latencies)

```
ping -s ftp.cdrom.com
64 bytes from 216.17.74.242: icmp_seq=20. time=92. ms
64 bytes from 216.17.74.242: icmp_seq=21. time=99. ms
64 bytes from 216.17.74.242: icmp_seq=22. time=89. ms
64 bytes from 216.17.74.242: icmp_seq=23. time=74. ms
```

This was to the New York City area around 1996

```
[L:\ACAD\CS481NET] ==> ping watson.ibm.com
PING watson.ibm.com: 56 data bytes
64 bytes from 129.34.139.4: icmp_seq=0. time=94. ms
64 bytes from 129.34.139.4: icmp_seq=1. time=62. ms
64 bytes from 129.34.139.4: icmp_seq=2. time=63. ms
```

That system doesn't exist in the DNS now.. so we pick another target which rejects our ping but politely notifies us that it did so.. Usually dropped pings just disappear.

```
acad/cs826 ==>  ping -s  -a www.nyu.edu
PING www.nyu.edu: 56 data bytes
ICMP  Communication  Administratively  Prohibited  from  gateway  WWHGWF-FDDI-0-
0.NYU.NET (128.122.253.70)
 for icmp from jmw (130.127.48.24) to WWWSERVER.NYU.EDU (128.122.108.9)
```

So we try pinging the router that rejected our ping..

```
acad/cs826 ==> ping -s 128.122.253.70
PING 128.122.253.70: 56 data bytes
64 bytes from WWHGWF-FDDI-0-0.NYU.NET (128.122.253.70): icmp_seq=0. time=45. ms
64 bytes from WWHGWF-FDDI-0-0.NYU.NET (128.122.253.70): icmp_seq=1. time=45. ms
64 bytes from WWHGWF-FDDI-0-0.NYU.NET (128.122.253.70): icmp_seq=2. time=52. ms
64 bytes from WWHGWF-FDDI-0-0.NYU.NET (128.122.253.70): icmp_seq=3. time=45. ms
```

The latency here seems about 25% better than it was in 1996

Various ping programs typically have options for specifying:

- The size of the ping packet (56–64) bytes
- The interval between ping requests (one second)
- Flood ping (requires root privliges)
- Solaris has a "UDP" ping mode to avoid ping filters!

```
acad/cs826 ==> ping -s -U www.nyu.com
PING www.nyu.com: 56 data bytes
92 bytes from ns.nylink.com (209.61.189.246): udp_port=33434. time=59. ms
92 bytes from ns.nylink.com (209.61.189.246): udp_port=33435. time=58. ms
92 bytes from ns.nylink.com (209.61.189.246): udp_port=33436. time=57. ms
92 bytes from ns.nylink.com (209.61.189.246): udp_port=33437. time=58. ms
```

The "last mile" latency puts us back near where we were in 1996.

Malicious use of ping – "smurfing" a famous DoS attack

- Forge the source address of your ping packet to be that of your victim
- Send pings to a subnet directed broadcast address 192.168.19.255
- If the subnet is "fully loaded" the victim gets 254 replies.
- Suppose a hacker flood pings at a rate of 40,000 bps on a dialup line
- The victim will receive ping responses at over 10,000,000 bps.. or whatever the bottleneck in the path to his site is!

A Sample "Ping" Program

Here are the values for protocol ... returned by getprotobyname();
These are defined in *in.h*

```
#define IPPROTO_IP      0    /* dummy for IP                 */
#define IPPROTO_ICMP    1    /* control message protocol     */
#define IPPROTO_GGP     3    /* gateway^2 (deprecated)       */
#define IPPROTO_TCP     6    /* tcp                          */
#define IPPROTO_EGP     8    /* exterior gateway protocol    */
#define IPPROTO_PUP    12    /* pup                          */
#define IPPROTO_UDP    17    /* user datagram protocol       */
#define IPPROTO_IDP    22    /* xns idp                      */

#define IPPROTO_RAW   255    /* raw IP packet                */
#define IPPROTO_MAX   256
```

Internet addresses required in *bind(), connect(), sendto(), recvfrom(), etc.* are specified as:
(These structures are also defined in in.h)

```
/*
 * Socket address, internet style.
 */

struct in_addr
{
      unsigned long s_addr;
};

struct sockaddr_in
{
       short      sin_family;
       unsigned short sin_port;
       struct in_addr sin_addr;
       char       sin_zero[8];
};
```

Structure used in high level interface to DNS lookup.. found in netdb.h

```
struct   hostent
{
   __const
   char  *h_name;                /* official name of host              */
   char  **h_aliases;            /* alias list                         */
   int   h_addrtype;             /* host address type                  */
   int   h_length;               /* length of address                  */
   char  **h_addr_list;          /* list of addresses from name server */
#define  h_addr   h_addr_list[0] /* address, for backward compatiblity */
};
```

```
/* ping.c  */

/* This is a sample "ping" program */

#include <stdio.h>
#include <errno.h>
#include <time.h>

#ifdef _POSIX_SOURCE
#include <unistd.h>
#endif

#ifdef __STDC__
#include <stdlib.h>
#endif

#include <string.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/socket.h>

struct timeval timeout_timeval;
struct timezone tz;
#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>

#include <netdb.h>

#ifndef FD_SET
#include <sys/select.h>
#endif
```

This is not a "standard" network header structure...
It is application specific to the ping program, and should not be confused with the unfortunately similarly named *struct hostent*

```
typedef struct host_entry {
     char                *hostname;          /* ascii host name             */
     struct sockaddr_in  saddr;              /* internet address            */
     int                 num_packets_sent;   /* number of ping packets sent */
     int                 i;
     struct timeval      last_time;          /* time of last packet sent */
} HOST_ENTRY;

HOST_ENTRY h;
int s;                       /* socket */
```

```
int main(int argc, char **argv)
{
  int c;
  char            *host;
  struct protoent *proto;
  struct hostent  *host_ent;
  struct in_addr  *host_add;
  static char     buffer[32];
  struct icmp     *icp = (struct icmp *) buffer;
  int             n,len;
```

Determine the protocol number to be used in creating the socket.
This number defines how what we write is to be interpreted....
In this case it is body of *icmp* message... Therefore *proto->p_proto = 1* here.

```
  if ((proto = getprotobyname("icmp")) == NULL)
       error_msg("icmp: unknown protocol");
```

Create the socket handle used to read and write data.
Parameters are

Protocol family	typically PF_UNIX or PF_INET	
Socket type	SOCK_DGRAM	==> UDP
	SOCK_STREAM	==> TCP
	SOCK_RAW	==> Low level (requires root priv.)
Protocol ID	Values defined in *in.h*	

```
  s = socket(PF_INET, SOCK_RAW, proto->p_proto);
  if (s < 0)
      error_msg("can't create raw socket");
```

Copy address of the name of the target host –– e.g. jmw.cs.clemson.edu
and call the name resolver which returns the address of a structure containing the IP address.

```
  host = argv[1];
  host_ent = gethostbyname(host);
```

Fill in the *sockaddr_in* structure. It specifies the dest address in *sendto()* calls.

```
  bcopy((char *)host_ent->h_addr, (char *)&h.saddr.sin_addr,
                  host_ent->h_length);
  h.saddr.sin_family = AF_INET;
```

Send a single ping and wait for reply.

```
  send_ping(s, &h);
  wait_for_reply();
}
```

The standard IP header checksum algorithm checks the headers of ICMP packets.

```
int in_cksum(u_short *p, int n)
{
  register u_short answer;
  register long sum = 0;
  u_short odd_byte = 0;

  while( n > 1 )  { sum += *p++; n -= 2; }

/* mop up an odd byte, if necessary */
  if( n == 1 ) {
      *(u_char *)(&odd_byte) = *(u_char *)p;
      sum += odd_byte;
  }

  sum = (sum >> 16) + (sum & 0xffff);    /* add hi 16 to low 16 */
  sum += (sum >> 16);                    /* add carry */
  answer = ~sum;                         /* ones-complement, truncate*/
  return (answer);
}
```

Build and send a single "ping" packet.

```
#define SIZE_ICMP_HDR 8
int send_ping(int s, HOST_ENTRY *h)
{
  static char buffer[32];
  struct icmp *icp = (struct icmp *) buffer;
  int         n,len;
```

Remember time of send in our host structure.

```
  gettimeofday(&h->last_time, &tz);
```

Fill in the icmp header..

```
  icp->icmp_type = ICMP_ECHO;
  icp->icmp_code = 0;
  icp->icmp_cksum = 0;
  icp->icmp_seq = h->i;
  icp->icmp_id =  0;   /* should use getpid() */
```

Despite how it might appear in this code... the ICMP checksum covers the *entire* message.

```
  len = SIZE_ICMP_HDR;
  icp->icmp_cksum = in_cksum( (u_short *)icp, len );
```

Send the ping packet. Parameters of send to are

- socket handle
- buffer address
- message length
- flags (almost always 0).
- pointer to destination address structure (would contain port # for UDP/TCP)
- length of the address structure.

```
  n = sendto(s, buffer, len, 0, (struct sockaddr *)&h->saddr,
                                  sizeof(struct sockaddr_in));
  printf("%d bytes transmitted \n", n);
}
```

```
#define DEFAULT_INTERVAL 25   /* default time between packets (msec) */
int interval = DEFAULT_INTERVAL;

int wait_for_reply()
{
int result;
static char        buffer[4096];
struct sockaddr_in response_addr;
struct ip          *ip;
int                hlen;
struct icmp        *icp;
int                n;
HOST_ENTRY         *h;

long              this_reply;
int               the_index;
struct timeval    sent_time;
int  ident = 0;

   result = recvfrom_wto(s, buffer, 4096,
                    (struct sockaddr *)&response_addr, interval);

   if (result < 0)
      return(0);
```

Although this program did *not* pass an IP header to *sendto(),*
it does receive the IP header from *recvfrom()!*

```
   ip = (struct ip *)buffer;
   hlen = ip->ip_hl << 2;
   if (result < hlen + ICMP_MINLEN)
       { return(1); /* too short */ }
```

Add IP header length to buffer address to set up pointer to icmp data
Then verify that type is correct and that ident matches that which was sent.

```
   icp = (struct icmp *)(buffer + hlen);

   if (( icp->icmp_type != ICMP_ECHOREPLY ) ||
       ( icp->icmp_id   != ident          ))
   {
       return (1); /* packet received, but not the one we are looking for! */
   }

   n = icp->icmp_seq;

}
```

Receive the ping response –– and any other ICMP junk floating around!

```
int recvfrom_wto(int s, char *buf, int len, struct sockaddr *saddr, int timo)
{
  int nfound,slen,n;
  struct timeval to;
  fd_set readset,writeset;
```

Setup time structure used to specify time out

```
  to.tv_sec  = timo/1000;
  to.tv_usec = (timo - (to.tv_sec*1000))*1000;
```

Initilialize handle sets used by select... Enable *only* the socket on which we expect the ICMP reply

```
  FD_ZERO(&readset);
  FD_ZERO(&writeset);
  FD_SET(s, &readset);
```

Wait until message received or timeout occur

```
  nfound = select(s + 1, &readset, &writeset, NULL, &to);
```

Check for error conditions and give up if they occur

```
  if (nfound < 0)
      error_msg("select error");
  if (nfound == 0)
      return -1;  /* timeout */
```

Packet can be read.. slen provides size of the buffer and returns len of amout read. Address of the sender is filled in the variable *saddr*

```
  slen = sizeof(struct sockaddr);
  n = recvfrom(s, buf, len, 0, saddr, &slen);
  if (n < 0)
      error_msg("recvfrom");

  printf("%d bytes received \n", n);
  return n;
}

error_msg(
char *msg)
{
   fprintf(stderr, "%s\n", msg);
   exit(1);
}
```

Augmenting Ping with IP options

IP options (sec 7.3)

Follow the IP header

Record route option

Laid out as follows:

Code = 7	Len = 39	Ptr = 4	IP addr1	IP addr2	IP addr3

Code = 7 ==> record route

Can capture a max of 9 hops since IP header has a max size of 60 bytes.

Record route is supported by the SUN/OS ping man page but doesn't work

Record route is not supported by OS/2 ping

Timestamp Option

Code 0x44	Len	Ptr	OF | Fl	TS1	TS2		
1	1	1	1	4	4		

OF is a counter of the number of timestamps that wouldn't fit

Fl – Flag

0 – Record timestamps only

1 – Record IP address and timestamp

3 – Insert timestamp only on IP address match

Why this protocol is unsatisfactory

Default units is ms since midnight UTC

But any can be used... just set high order bit

Clocks may not be synched at all!

Indirectly determining bandwidth and delay with PING

H_0 −−−−−−−−−−−> H_1 −−−−−−−−−−−−−−−> H_2

Suppose we wish to determine the Bandwidth *B* and the latency *L* between H_1 and H_2

We construct two test packets

P_1 has length J
P_2 has length K and J < K

$t_{1,j}$ = round trip time for P_1 to H_1
$t_{2,j}$ = round trip time for P_1 to H_2

$t_{1,k}$ = round trip time for P_2 to H_1
$t_{2,k}$ = round trip time for P_2 to H_2

$\Delta t_j = t_{2,j} - t_{1,j} \quad = 2\ J / B + 2L$

$\Delta t_k = t_{2,k} - t_{1,k} \quad = 2\ K / B + 2L$

Thus,

$\Delta t_k - \Delta t_j = 2\ (K - J) / B$
or
$B = 2\ (K - J) / (\Delta t_k - \Delta t_j)$
and
$L = \Delta t_k / 2 - K / B = \Delta t_j / 2 - J / B$

1408 bytes from 192.168.1.1: icmp_seq=0 ttl=254 time=35.3 ms
1408 bytes from 192.168.1.1: icmp_seq=1 ttl=254 time=34.9 ms

72 bytes from 192.168.1.1: icmp_seq=0 ttl=254 time=3.6 ms
72 bytes from 192.168.1.1: icmp_seq=1 ttl=254 time=3.2 ms

1408 bytes from 130.127.48.190: icmp_seq=0 ttl=255 time=7.4 ms
1408 bytes from 130.127.48.190: icmp_seq=1 ttl=255 time=7.3 ms

72 bytes from 130.127.48.190: icmp_seq=0 ttl=255 time=0.9 ms
72 bytes from 130.127.48.190: icmp_seq=1 ttl=255 time=0.9 ms

Here B = 1000 * 2 * 8 (1408−72) / ((35.1 − 7.35)− (3.4 − 0.9)) = 846.6 Kbps
L = (35.1 −7.35) / 2 − 8 * 1408 / 846.6 = 0.57 msec