CPS 470/570: Computer Networks

Assignment 4, due 11:55 PM, 4-19-2017

Receive an *F* for this course if dishonesty occurs

Receive 5 bonus points if submit it without errors one day before the deadline

1. Purpose

Implement the *sequential* version of traceroute. **Receive 10 bonus points** if you implement the ***parallel*** version of traceroute.

2. Description

2.1. Run-Time Issues

Several issues you should pay attention to. First, to open Visual Studio, you may go to start/Visual Studio 2013, and then **right click** Visual Studio 2013. You should then choose "**Run as administrator**." This allows you to use raw sockets in your program. Second, you should **disable your anti-virus software** when testing this program. After testing, make sure to enable your anti-virus software. Finally, before testing you should **enable ICMP echo requests on your computer**. You may google "enable ICMP echo requests Your_Operating_System_Name" to find how to fix this. The following works for Windows 7.

To **enable ICMP echo requests** for this assignment, go to Control Panel/System and Security/Windows Firewall/Advanced Settings, and then click *Inbound Rules*. Sort Inbound Rules by Name, find *ICMP requests* and **right click** ICMP requests, and choose **Enable Rule**. You have to **restart** your computer afterwards. Again, **make sure** that turn off your antivirus software (e.g., Symantec) on your computer when you test your program.

Make sure you use large packet size for your packets; otherwise routers might consider your packets as malicious ones.

2.2. Overview

Traceroute operates by sending a sequence of probes (i.e., packets) towards a given destination *D*. As these probes work their way toward to *D*, they pass through a series of routers. A preset TTL (time-to-live) value (e.g., TTL=5) is carried in the packet. Every time the packet is forwarded from a router to its adjacent router, the TTL value is decremented by 1. The packet is forwarded in the network until TTL reaches zero.

In your implementation, each probe *i* has the TTL value set to *i*, which causes router *i* along the path to discard the packet and generate/return a "TTL expired" message (see Section 2.4 below). By successively setting TTL from 1 to *N*, where *N* is the number of hops in the path, traceroute obtains the IP addresses of each router. Performing reverse DNS lookups on these addresses, traceroute also prints the DNS names of these routers.

Sample execution of your Traceroute:

```
C:\> trace.exe www.yahoo.com
Tracerouting to 66.94.230.52...
```

```
 1  dc (128.194.135.65)  0.226 ms
 2  <no DNS entry> (128.194.135.62)  0.735 ms (1)
 3  hrbb-1-hrbb-nb-e-8.net.tamu.edu (165.91.133.25)  0.611 ms (1)
 4  <no DNS entry> (10.3.3.105)  0.610 ms (1)
 5  <no DNS entry> (10.3.3.57)  0.734 ms (2)
 6  csce-7--rngm-ci-e-3.net.tamu.edu (165.91.2.3)  1.110 ms (1)
 7  tamu-gw-f1-1-0.tx-bb.net (165.91.254.6)  2.859 ms (1)
 8  hou-core-at-1-0-0-3.tx-bb.net (192.12.10.73)  4.737 ms (1)
 9  aus-core-at-1-0-1-1.tx-bb.net (192.12.10.69)  10.107 ms (1)
10  sbis-gw-fa8-0-0.tx-bb.net (192.12.10.38)  9.731 ms (1)
11  <no DNS entry> (151.164.21.245)  13.604 ms (1)
12  bb1-g1-0.austtx.sbcglobal.net (151.164.20.225)  15.985 ms (1)
13  bb1-p5-0.hstntx.sbcglobal.net (151.164.242.245)  27.983 ms (1)
14  bb2-p14-0.hstntx.sbcglobal.net (151.164.240.242)  41.483 ms (1)
15  core2-p6-0.crhstx.sbcglobal.net (151.164.188.9)  17.485 ms (1)
16  core1-p11-0.cratga.sbcglobal.net (151.164.240.114)  38.983 ms (1)
17  core2-p1-0.cratga.sbcglobal.net (151.164.241.82)  40.479 ms (3)
18  core2-p11-0.crhnva.sbcglobal.net (151.164.241.93)  41.484 ms (1)
19  bb2-p4-0.hrndva.sbcglobal.net (151.164.191.102)  41.713 ms (1)
20  ex2-p5-0.eqabva.sbcglobal.net (151.164.191.138)  41.713 ms (1)
21  asn10310-yahoo.eqabva.sbcglobal.net (151.164.249.118)  94.056 ms (1)
22  vl149.pat2.pao.yahoo.com (216.115.96.32)  91.555 ms  (1)
23  vl35.bas1-m.scd.yahoo.com (66.218.82.197)  97.174 ms (1)
24  unknown-66-218-82-230.yahoo.com (66.218.82.230)  90.174 ms (1)
25  p21.www.scd.yahoo.com (66.94.230.52)  84.435 ms (1)

Total execution time: 650 ms
```

Columns from left to right refer to: 1) hop number (i.e., TTL value); 2) DNS name of the router at that TTL; 3) IP address of the router; 4) one RTT measurement; 5) how many probes were sent with this value of TTL. The reason for having multiple probes is that some of them may be lost and a retransmission may be required.

2.3. ICMP Sockets

Traceroute is implemented with ICMP messages (see textbook). The following instructions help you send/receive ICMP packets to destination *D*.

In order to send and receive ICMP packets, you will need an ICMP socket:

```
/* ready to create a socket */
sock = socket (AF_INET, SOCK_RAW, IPPROTO_ICMP);

if (sock == INVALID_SOCKET)
{
      printf ("Unable to create a raw socket: %d\n", WSAGetLastError ());
      // do some cleanup
      ...
      // then exit
      exit(-1);
}
```

After you create a raw socket, **no need to bind** port number to the socket. **No need to call function connect()**. Call the functions **sendto(), recvfrom()** in the **same** way that you used for a UDP socket in Assignment #2 DNS resolver.

To ensure proper structure packing, use the following code:

```
#define IP_HDR_SIZE           20    /* RFC 791 */
#define ICMP_HDR_SIZE         8     /* RFC 792 */
/* payload size of an ICMP message originated in the program */
#define MAX_SIZE              65200
/* the size of the whole IP datagram */
#define MAX_ICMP_SIZE         (MAX_SIZE + ICMP_HDR_SIZE)
/* the returned ICMP message will most likely include only 8 bytes
* of the original message plus the IP header (as per RFC 792); however,
* longer replies (e.g., 68 bytes) are possible */
#define MAX_REPLY_SIZE        (IP_HDR_SIZE + ICMP_HDR_SIZE + MAX_ICMP_SIZE)

/* ICMP packet types */
#define ICMP_ECHO_REPLY       0
#define ICMP_DEST_UNREACH     3
#define ICMP_TTL_EXPIRE       11
#define ICMP_ECHO_REQUEST     8

#ifdef _MSC_VER
/* remember the current packing state */
#pragma pack (push, traceroute)
/* use forceful structure packing into bytes in MS C++; Borland C++ is OK */
#pragma pack (1)
#endif

/* define the IP header (20 bytes) */
class IPHeader {
public:
  u_char h_len:4;       /* 4 bits: length of the header in dwords */
  u_char version:4;     /* 4 bits: version of IP, i.e., 4 */
  u_char tos;           /* type of service (TOS) */
  u_short len;          /* length of packet in dwords */
  u_short ident;        /* unique identifier */
  u_short flags;        /* flags together with fragment offset - 16 bits */
  u_char ttl;           /* time to live */
  u_char proto;         /* protocol number (6-TCP, 17-UDP, etc.) */
  u_short checksum;     /* IP header checksum */
  u_long source_ip;
  u_long dest_ip;
};

/* define the ICMP header (8 bytes) */
class ICMPHeader{
public:
  u_char type;                /* ICMP packet type */
  u_char code;                /* type subcode */
  u_short checksum;           /* checksum of the ICMP */
  u_short id;                 /* application-specific ID */
  u_short seq;                /* application-specific sequence */
};

#ifdef _MSC_VER
/* now restore the previous packing state */
#pragma pack (pop, traceroute)
#endif
```

The structure of the ICMP header is shown in Figure 1. The first two fields are used to signal which type of ICMP message is carried in the packet (see textbook for the various values). The ID and Sequence Number fields should be used to match router responses to the transmitted packets (see below).

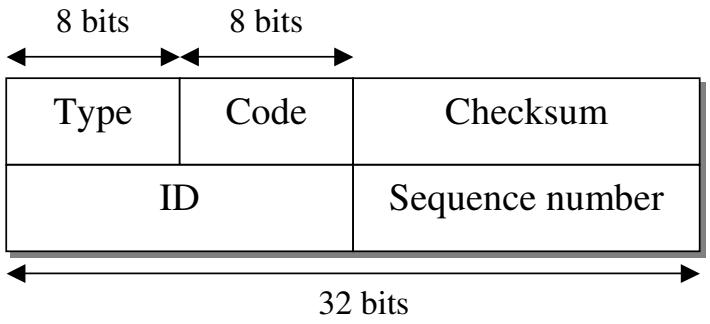

Figure 1. ICMP header (8 bytes).

Checksum code is given below:

```
/*
* =====================================================================
* ip_checksum: compute ICMP header checksum.
*
* Returns the checksum. No errors possible.
*
* =====================================================================
*/
u_short ip_checksum (u_short * buffer, int size)
{
      u_long cksum = 0;

      /* sum all the words together, adding the final byte if size is odd */
      while (size > 1)
      {
            cksum += *buffer++;
            size -= sizeof (u_short);
      }

      if (size)
            cksum += *(u_char *) buffer;

      /* do a little shuffling */
      cksum = (cksum >> 16) + (cksum & 0xffff);
      cksum += (cksum >> 16);

      /* return the bitwise complement of the resulting mishmash */
      return (u_short) (~cksum);
}
```

2.4. Transmitting an ICMP Packet

Your code will use ICMP echo request packets. Your code will terminate when you receive an ICMP echo response from the end host rather than an ICMP port unreachable. See textbook for ICMP message types.

Sample code below shows how to use the above classes to transmit a ping message towards a destination.

```
// buffer for the ICMP header
u_char send_buf[MAX_ICMP_SIZE];     /* IP header is not present here */

class ICMPHeader *icmp = (ICMPHeader *) send_buf;

// set up the echo request
// no need to flip the byte order since fields are 1 byte each
icmp->type = ICMP_ECHO_REQUEST;
icmp->code = 0;

// set up optional fields as needed
...
// initialize checksum to zero
icmp->checksum = 0;

/* calculate the checksum */
int packet_size = sizeof(ICMPHeader);           // 8 bytes
icmp->checksum = ip_checksum ((u_short *) send_buf, packet_size);

// set proper TTL
int ttl = <desired TTL>;
// need Ws2tcpip.h for IP_TTL
if (setsockopt (sock, IPPROTO_IP, IP_TTL, (const char *) &ttl,
             sizeof (ttl)) == SOCKET_ERROR) {
      perror ("setsockopt failed\n");
      closesocket (sock);
      // some cleanup
      exit(-1);
}

// use regular sendto on the above socket
...
```

2.5. Receiving an ICMP Packet

Once the router discards your packet based on expired TTL, it copies the first 28 bytes (starting with the IP header) of that packet and sends them back to your host using an ICMP "TTL Expired" message. The format of this message is shown in Figure 2, where the first 28 bytes are generated by the router and the remaining 28 bytes are from your original packet.[1] Notice in the figure, that the entire IP packet is delivered to the socket. Parsing the *first* IP header, you can obtain the router's IP address.

[1] Some routers return more than 28 bytes of your packet, so be prepared to experience longer return packets. In addition, you must use the header length field in the IP header to be able to process IP headers longer than 20 bytes.

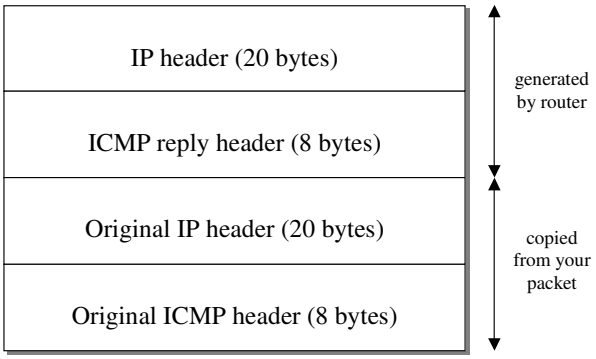

Figure 2. "TTL Expired" packet format (standard is 56 bytes).

Since there are no port numbers in ICMP, every received ICMP packet is delivered to *all open ICMP sockets*! Thus, it is sometimes possible that your socket will receive unrelated ICMP traffic, which you should ignore. To check if this packet was in response to your traceroute probes, set the ID field of all outgoing packets to your process ID and check whether the ID field of the returned *original* ICMP header matches the ID of your process:

```
// set up optional fields as needed
icmp->ID = (u_short) GetCurrentProcessId ();
// initialize checksum to zero
icmp->checksum = 0;
// compute checksum and transmit the packet
```

To parse the returned message, allocate enough memory to receive the packet and assign pointers to each field (note that this code does not handle variable-size IP headers):

```
u_char rec_buf[MAX_REPLY_SIZE];/* this buffer starts with an IP header */
IPHeader *router_ip_hdr = (IPHeader *)rec_buf;
ICMPHeader *router_icmp_hdr = (ICMPHeader *)(router_ip_hdr + 1);
IPHeader *orig_ip_hdr = (IPHeader *)(router_icmp_hdr + 1);
ICMPHeader *orig_icmp_hdr = (ICMPHeader *)(orig_ip_hdr + 1);
// receive from the socket into buffer rec_buf
...
// check to see if this is TTL_expired
if (router_icmp_hdr->type == <value> && router_icmp_hdr->code == <value>)
{
      // check if process ID matches
      if (orig_icmp_hdr->ID == GetCurrentProcessId())
      {
            // take router_ip_hdr->source_ip and
            // perform a DNS lookup, then report to the user
      }
      // else ignore the message
}
```

2.6. Parallel Version (10 bonus points for CPS 470/570)

While this homework **requires a sequential version** of traceroute (send one probe, get one response), your program can send all probes at once (i.e., **in parallel**) to routers **with a single thread**. Using a common assumption that the maximum distance to any destination is 30 hops, you should send all 30 probes simultaneously (each with a different TTL) and then wait for

responses from the routers. In order to know which router sent which response, use the *sequence* field in the ICMP header to encode the TTL. In case you hear nothing from a router at a certain TTL *x*, retransmit the probe for that particular TTL after a timeout (experiment with several values between 0.1 and 3 seconds). If the router at hop *x* does not respond after 3 attempts, print "<ICMP timeout>" next to hop *x*:

```
10  sbis-gw-fa8-0-0.tx-bb.net (192.12.10.38)  9.731 ms (1)
11  <no DNS entry> (151.164.21.245)  13.604 ms (1)
12  <ICMP timeout>
13  bb1-p5-0.hstntx.sbcglobal.net (151.164.242.245)  27.983 ms (1)
```

If the target end-host responds to ICMP ping messages, all packets with a TTL larger than the distance to the host will result in ICMP echo responses. Thus, your code should check for both "TTL expired" and "echo reply" messages coming from the network.

2.7. DNS

Your program then maps the IP addresses of routers into their DNS names. For reporting **DNS names** of routers, send multiple reverse DNS lookup requests from a single thread. You can use `gethostbyaddr()` or `getnameinfo()`, but this will slow down your execution since `gethostbyaddr` is blocking and only one DNS request can be pending at any time.

2.8. Report

Write about your implementation and analyze the performance of your code. This includes how long it takes to trace certain paths, how many packets (including retransmissions) are usually transmitted by your program, average packet loss observed during traceroute, what timeout is best for an average Internet path, etc.