CSCE 463/612: Networks and Distributed Processing

Homework 4 (100 pts)

Due date: 5/4/25

1. Purpose

Implement a fast version of traceroute and learn about the topology of the Internet.

2. Description

Traceroute operates by sending a sequence of probes towards a given destination *D*. Each probe *i* has the TTL value set to *i*, which causes router *i* along the path to discard the packet and generate a "TTL expired" message. By iterating through TTL values 1, 2, …, *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 corresponding DNS names. In this homework, your traceroute should be optimized to send all probes at once (i.e., in parallel) instead of sequentially. This allows it to complete much faster than the regular version.

2.1. Code (75 pts)

The program must accept a single destination (hostname or IP) to which to perform the trace and then produce output in the following format:

```
C:\> trace.exe www.yahoo.com
Tracerouting to 66.94.230.52...
 1  dc (128.194.135.65)  0.226 ms (1)
 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  *
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) the 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/ignored and a retransmission may be required.

Your program must be *single-threaded* (see the exceptions below) and able to perform parallel traceroutes to targets that can be input as both IPs and hostnames from the command line. To achieve *microsecond* RTT resolution shown above, see:

http://msdn.microsoft.com/en-us/library/windows/desktop/ms644905(v=vs.85).aspx
http://msdn.microsoft.com/en-us/library/windows/desktop/ms644904(v=vs.85).aspx

You should limit the number of probes per hop to three (after which the hop gets an asterisk) and design an algorithm for dynamically setting per-hop timeouts. To decide the proper RTO (retransmission timeout) for each hop, monitor packets from routers with larger TTL and use their RTTs to ballpark the timeout for the missing packet. For example, assume you start with default timeouts of 500 ms for all hops and transmit 30 probes at time 0. Then, suppose hop 5 does not respond, but hop 4 returns a reply within 30 ms and hop 6 within 40. Then, you might argue that adjusting the initial RTT of hop 5 to 70 ms (i.e., double the average of hop 4 and 6) is reasonable. Explain in the report how your algorithm works and how it handles cases when a) hop 4 responds but 6 doesn't or vice versa; b) neither of the adjacent hops responds. To manage $N = 30$ timeouts, store the time of a future retx event for each outstanding packet in a min-heap and allow for this value to be dynamically updated.

DNS lookups must be transmitted to your local DNS server *as soon as* the corresponding IP is obtained from the "TTL Expired" message. Adapt your hw#2 code to allow asynchronous operation, i.e., not block waiting for a response. You do not need to retransmit lost DNS requests and should timeout all outstanding DNS queries after 5 seconds. You can use select with your DNS and ICMP sockets (or `WSAEventSelect` with `WaitForMultipleObjects`), where the timeout is based on the soonest retx event. *If you homework #2 does not work, you can use N parallel threads and gethostbyaddr() within each thread. However, screen printouts must still be in order.*

Finally, your code should detect other ICMP errors and print their code/type next to each router (e.g., "other error: code 3, type 1"). See

http://en.wikipedia.org/wiki/Internet_Control_Message_Protocol

for more details.

2.2. Report (25 pts)

For the report, use a list of IPs from hw1 (crawled hosts) to extract 10K random IPs that respond to ICMP ping. Then, perform traceroutes to these destinations (without DNS lookups) and answer the following questions:

1. Show a trace from the longest path with a responsive destination. Discuss if any of the targets in your list are reachable at distance beyond 30 hops.
2. Plot the distribution of hop count to all 10K targets using a histogram.
3. What is the total number of router IPs you found and how many of them are unique?

4. Show the distribution of delay needed to trace a path using a histogram with bin size 50 ms.
5. Suggest a design for the batch-mode portion of this homework that avoids repeatedly hitting the nearby routers. Observe that for 10K destinations, your default program will elicit ICMP errors from each TAMU router (i.e., the first 5-6 hops) 10K times. This is not only redundant, but also suspicious (e.g., possibly indicative of a DoS attack on the routers, abnormal host configuration, or hostile network exploration). The goal is to create an alternative mechanism that would allow you to dynamically construct a map of the Internet topology, detect which routers have been seen before, and avoid hitting them with unnecessary traffic. *Hint: aim to minimize the total number of transmitted packets, which should automatically achieve optimal redundancy.*

2.3. Extra Credit (20 pts)

Implement the design in question 5 of the report.

3. Details

3.1. ICMP Sockets

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 an ICMP socket: error %d\n", WSAGetLastError ());
       exit(-1);
}
```

You can use `sendto()/ recvfrom()` as with UDP, but no port binding is required/allowed. To ensure proper struct packing, use the following code:

```
#define IP_HDR_SIZE           20      // RFC 791
#define ICMP_HDR_SIZE         8       // RFC 792
// max payload size of an ICMP message originated in the program
#define MAX_SIZE              65200
// max size of an 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_EXPIRED      11
#define ICMP_ECHO_REQUEST     8

// remember the current packing state
#pragma pack (push)
#pragma pack (1)

// define the IP header (20 bytes)
struct IPHeader {
  u_char h_len:4;      // lower 4 bits: length of the header in dwords
  u_char version:4;    // upper 4 bits: version of IP, i.e., 4
  u_char tos;          // type of service (TOS), ignore
  u_short len;         // length of packet
```

```
  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)
struct ICMPHeader{
  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
};

// now restore the previous packing state
#pragma pack (pop)
```

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 lecture slides for the various values). The ID and Sequence fields should be used to match router responses to the transmitted packets (see below).

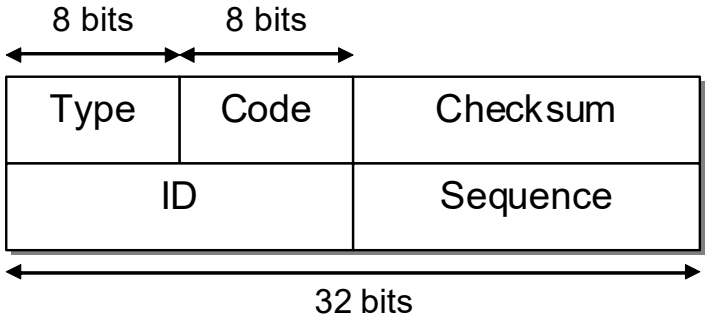

Figure 1. ICMP header (8 bytes).

The ICMP checksum runs over the entire packet (starting from the ICMP header) and is identical to UDP/TCP/IP checksum. Its general code is given by the following:

```
// ====================================================================
// ip_checksum: compute Internet checksums
// 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;
```

```
        // add carry bits to lower u_short word
        cksum = (cksum >> 16) + (cksum & 0xffff);

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

3.2. Transmitting ICMP Packets

While regular traceroute operates using UDP on some unused port, your code will use ICMP echo-request packets (i.e., ping) and will terminate when you receive an ICMP echo response (i.e., ping reply) from the end host.

The 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

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 ID/SEQ 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 = ...
// need Ws2tcpip.h for IP_TTL, which is equal to 4; there is another constant with the same
// name in multicast headers - do not use it!
if (setsockopt (sock, IPPROTO_IP, IP_TTL, (const char *) &ttl, sizeof (ttl)) == SOCKET_ERROR)
{
        printf ("setsockopt failed with %d\n", WSAGetLastError());
        closesocket (sock);
        // some cleanup
        exit(-1);
}

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

3.3. Receiving ICMP Packets

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[1] are generated by the router and the remaining 28 bytes are from your original packet. Parsing the *first* IP header, you can obtain the router's IP address.

[1] The IP header can be longer than 20 bytes due to options. Use the *header length* field to detect these cases.

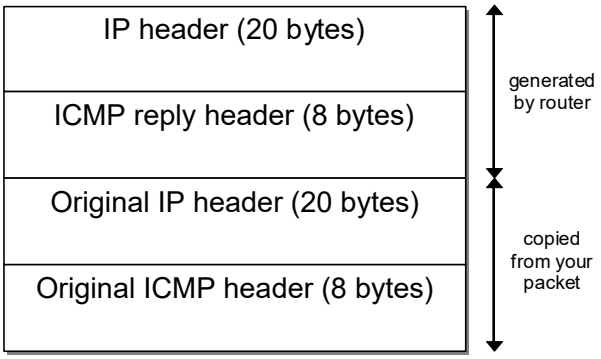

Figure 2. "TTL Expired" packet format (standard is 56 bytes).

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 rec_buf
...
// check if this is TTL_expired; make sure packet size >= 56 bytes
if (router_icmp_hdr->type == ... && router_icmp_hdr->code == ...)
{
        if (orig_ip_hdr->protocol == ICMP)
        {
                // check if process ID matches
                if (orig_icmp_hdr->ID == GetCurrentProcessId())
                {
                        // take router_ip_hdr->source_ip and
                        // initiate a DNS lookup
                }
        }
}
```

Sometimes routers may return additional information following the four headers in Figure 2, which you can safely ignore. However, you should be prepared to see packets larger than 56 bytes (168 is another common size).

ICMP echo replies from the destination have a different format, in which only the first two headers of Figure 2 are present and the total packet size is 28 bytes. In this case, you can find your SEQ and ID fields in the ICMP reply header. Modify the pseudocode above accordingly.

3.4. Parallel Version

While it is trivial to implement a sequential version of traceroute (send one probe, get one response), this homework requires that probes be sent in parallel to all routers *from 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 the 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. If the router at hop *x* does not respond after 3 attempts, print * next to it:

```
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  *
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 truncate the printout at the *first* hop that returns an echo reply.

3.5. Permissions, UAC, and Firewall Issues

Raw sockets require administrator privileges on the host. In addition, the latest versions of Windows require that you *completely disable* UAC (user account control) and reboot (Control Panel → User Accounts → Change User Account Control Settings). Otherwise, you need to run Visual Studio in admin mode each time (i.e., right click and select "Run as Administrator").

The Windows firewall is by default configured to block incoming "TTL Expired" ICMP packets from reaching your program. To overcome this, you can disable the firewall (not recommended) or simply configure a new *inbound* rule under Windows Firewall with Advanced Security (Control Panel→Administrative Tools) to allow all ICMP packets (New Rule→Custom→All Programs, select Protocol Type "ICMPv4"). It is generally recommended that you enable the firewall to pass *all* ICMP codes (not just "TTL Expired") as some routers may return host or network unreachable errors, which can be useful for stopping the trace quicker.

3.6. Large-Scale Measurement

The batch-mode algorithm may look something like this:

```
DWORD WINAPI TraceThread (LPVOID params)
{
        ...
        do{
                // grab from a producer-consumer queue
                ip = GetNextIP();
                if (ping(ip) == SUCCESS)      // ping the IP with TTL 30
                {
                        // traceroute to ip using TTL [1, 2, …, 30]; record statistics
                        if (successful traceroute)
                                InterlockedIncrement(&success);
                }
        } while (success < 10K);
}
```

Note that in batch mode you do not need to perform any DNS lookups. Furthermore, to keep things rolling, you can set a hard upper bound on how long each trace should take, e.g., 2 seconds. Finally, ISPs and routers will be rate-limiting ICMP responses to your traffic and/or raising anomaly alarms based on the volume of errors generated by your host, which means that running many concurrent threads may exacerbate this problem and actually hurt performance. It is thus recommended that you stay limited to a handful of threads (i.e., 5-10).

463/612 Homework 4

Name: _____

Function	Points	Break down	Item	Deduction
Printouts	45	5	Hop distance incorrect or not displayed	
		5	Doesn't report router DNS name	
		5	Non-concurrent DNS lookups	
		5	Doesn't report router IP	
		5	Fails to print the RTT for each hop	
		5	Non-microsecond resolution for delays	
		5	Fails to print number of probes sent/hop	
		5	Total execution time incorrect	
		5	Fails to parse/display other ICMP errors	
Operation	30	15	Routers are incorrect for the tested path (e.g., bogus, not in order, extra hops)	
		5	Timeouts not dynamically adaptable	
		5	Retransmission doesn't work	
		5	Can't parse variable-sized IP headers	
Misc				
Report	25	5	Longest trace	
		5	Distribution of hop count	
		5	Total IPs found and how many unique	
		5	Distribution of delay per path	
		5	Efficient design for batch mode	

Total points: _____