Winsock Programmer's FAQ
Examples: Ping: Raw Sockets Method

This pinger uses "raw sockets", so it requires Winsock 2. It's about 3 times longer (288 non-comment lines versus 98) than the ICMP method, but it will continue to work, while the ICMP method might fail to work on future versions of Windows. This program is also much more flexible than the ICMP.DLL pinger.

To use this program on Windows NT derivatives, you must be logged in as an Administrator.

This program is split into two major parts: a driver part and a "pinger" part. The driver mainly just declares `main()`, which calls the functions in the pinger part in the proper sequence. The pinger part is mostly reusable as-is, although you will probably want to do things like exchanging the output statements for encoded return values. There is also a separate module for the IP checksum calculation function because it is not specifically tied to pinging; this same algorithm is used in other parts of TCP/IP.

This program allows you to change the ICMP packet's TTL (time to live) value. From this, you can do several other interesting things, like developing a traceroute utility, and finding the next hop (such as a router) on your network. Notice that we don't use the "`ttl`" field in `struct IPHeader`, because we can only *receive* the full IP header, not send it. Instead, we use the `setsockopt()` with the `IP_TTL` flag to set the TTL option in the IP header.

You might want to add timeout functionality to this program, so that it doesn't wait forever for a ping reply. Two ways to handle this are to 1) spin the ping off into a separate thread and handle the timeout from the main thread; or 2) drop in a call to `select()` before the `recvfrom()` call, passing something reasonable for the *timeout* argument.

This program is based on a program in the Win32 SDK, though hardly any of the original code remains. Also, this version is a bit smarter, compiles under both Microsoft and Borland C++, and should be much easier to understand and reuse.

---

rawping_driver.cpp

```
/**********************************************************************
 rawping_driver.cpp - A driver program to test the rawping.cpp module.

 Building under Microsoft C++ 5.0:

    cl -GX rawping.cpp rawping_driver.cpp ip_checksum.cpp ws2_32.lib

 Building under Borland C++ 5.0:
```

```
    bcc32 rawping.cpp rawping_driver.cpp ip_checksum.cpp ws2_32.lib

 ----------------------------------------------------------------------
 Change log:
    9/21/1998 - Added TTL support.

    2/14/1998 - Polished the program up and separated out the
        rawping.cpp and ip_checksum.cpp modules.  Also got it to work
        under Borland C++.

    2/12/1998 - Fixed a problem with the checksum calculation.  Program
        works now.

    2/6/1998 - Created using Microsoft's "raw ping" sample in the Win32
        SDK as a model.  Not much remains of the original code.
**********************************************************************/

#include <winsock2.h>
#include <iostream.h>

#include "rawping.h"

#define DEFAULT_PACKET_SIZE 32
#define DEFAULT_TTL 30
#define MAX_PING_DATA_SIZE 1024
#define MAX_PING_PACKET_SIZE (MAX_PING_DATA_SIZE + sizeof(IPHeader))

int allocate_buffers(ICMPHeader*& send_buf, IPHeader*& recv_buf,
        int packet_size);


//////////////////////////////////////////////////////////////////////
// Program entry point

int main(int argc, char* argv[])
{
    // Init some variables at top, so they aren't skipped by the
    // cleanup routines.
    int seq_no = 0;
    ICMPHeader* send_buf = 0;
    IPHeader* recv_buf = 0;

    // Did user pass enough parameters?
    if (argc < 2) {
        cerr << "usage: " << argv[0] << " <host> [data_size] [ttl]" <<
                endl;
        cerr << "\tdata_size can be up to " << MAX_PING_DATA_SIZE <<
                " bytes.  Default is " << DEFAULT_PACKET_SIZE << "." <<
                endl;
        cerr << "\tttl should be 255 or lower.  Default is " <<
                DEFAULT_TTL << "." << endl;
        return 1;
    }

    // Figure out how big to make the ping packet
    int packet_size = DEFAULT_PACKET_SIZE;
    int ttl = DEFAULT_TTL;
    if (argc > 2) {
        int temp = atoi(argv[2]);
        if (temp != 0) {
            packet_size = temp;
```

```
        }
        if (argc > 3) {
            temp = atoi(argv[3]);
            if ((temp >= 0) && (temp <= 255)) {
                ttl = temp;
            }
        }
    }
    packet_size = max(sizeof(ICMPHeader),
            min(MAX_PING_DATA_SIZE, (unsigned int)packet_size));

    // Start Winsock up
    WSAData wsaData;
    if (WSAStartup(MAKEWORD(2, 1), &wsaData) != 0) {
        cerr << "Failed to find Winsock 2.1 or better." << endl;
        return 1;
    }

    // Set up for pinging
    SOCKET sd;
    sockaddr_in dest, source;
    if (setup_for_ping(argv[1], ttl, sd, dest) < 0) {
        goto cleanup;
    }
    if (allocate_buffers(send_buf, recv_buf, packet_size) < 0) {
        goto cleanup;
    }
    init_ping_packet(send_buf, packet_size, seq_no);

    // Send the ping and receive the reply
    if (send_ping(sd, dest, send_buf, packet_size) >= 0) {
        while (1) {
            // Receive replies until we either get a successful read,
            // or a fatal error occurs.
            if (recv_ping(sd, source, recv_buf, MAX_PING_PACKET_SIZE) <
                    0) {
                // Pull the sequence number out of the ICMP header.  If
                // it's bad, we just complain, but otherwise we take
                // off, because the read failed for some reason.
                unsigned short header_len = recv_buf->h_len * 4;
                ICMPHeader* icmphdr = (ICMPHeader*)
                        ((char*)recv_buf + header_len);
                if (icmphdr->seq != seq_no) {
                    cerr << "bad sequence number!" << endl;
                    continue;
                }
                else {
                    break;
                }
            }
            if (decode_reply(recv_buf, packet_size, &source) != -2) {
                // Success or fatal error (as opposed to a minor error)
                // so take off.
                break;
            }
        }
    }

cleanup:
    delete[]send_buf;
    delete[]recv_buf;
```

```
    WSACleanup();
    return 0;
}


//////////////////////////// allocate_buffers ////////////////////////////
// Allocates send and receive buffers.  Returns < 0 for failure.

int allocate_buffers(ICMPHeader*& send_buf, IPHeader*& recv_buf,
        int packet_size)
{
    // First the send buffer
    send_buf = (ICMPHeader*)new char[packet_size];
    if (send_buf == 0) {
        cerr << "Failed to allocate output buffer." << endl;
        return -1;
    }

    // And then the receive buffer
    recv_buf = (IPHeader*)new char[MAX_PING_PACKET_SIZE];
    if (recv_buf == 0) {
        cerr << "Failed to allocate output buffer." << endl;
        return -1;
    }

    return 0;
}
```

---

rawping.cpp

```
/**************************************************************************
 rawping.cpp - Contains all of the functions essential to sending "ping"
    packets using Winsock 2 raw sockets.  Depends on ip_checksum.cpp for
    calculating IP-style checksums on blocks of data, however.
**************************************************************************/

#include <winsock2.h>
#include <ws2tcpip.h>
#include <iostream.h>

#include "rawping.h"
#include "ip_checksum.h"


//////////////////////////// setup_for_ping ////////////////////////////
// Creates the Winsock structures necessary for sending and recieving
// ping packets.  host can be either a dotted-quad IP address, or a
// host name.  ttl is the time to live (a.k.a. number of hops) for the
// packet.  The other two parameters are outputs from the function.
// Returns < 0 for failure.

int setup_for_ping(char* host, int ttl, SOCKET& sd, sockaddr_in& dest)
{
    // Create the socket
    sd = WSASocket(AF_INET, SOCK_RAW, IPPROTO_ICMP, 0, 0, 0);
    if (sd == INVALID_SOCKET) {
        cerr << "Failed to create raw socket: " << WSAGetLastError() <<
                endl;
```

```
        return -1;
    }

    if (setsockopt(sd, IPPROTO_IP, IP_TTL, (const char*)&ttl,
            sizeof(ttl)) == SOCKET_ERROR) {
        cerr << "TTL setsockopt failed: " << WSAGetLastError() << endl;
        return -1;
    }

    // Initialize the destination host info block
    memset(&dest, 0, sizeof(dest));

    // Turn first passed parameter into an IP address to ping
    unsigned int addr = inet_addr(host);
    if (addr != INADDR_NONE) {
        // It was a dotted quad number, so save result
        dest.sin_addr.s_addr = addr;
        dest.sin_family = AF_INET;
    }
    else {
        // Not in dotted quad form, so try and look it up
        hostent* hp = gethostbyname(host);
        if (hp != 0) {
            // Found an address for that host, so save it
            memcpy(&(dest.sin_addr), hp->h_addr, hp->h_length);
            dest.sin_family = hp->h_addrtype;
        }
        else {
            // Not a recognized hostname either!
            cerr << "Failed to resolve " << host << endl;
            return -1;
        }
    }

    return 0;
}



//////////////////////////// init_ping_packet ////////////////////////////
// Fill in the fields and data area of an ICMP packet, making it
// packet_size bytes by padding it with a byte pattern, and giving it
// the given sequence number.  That completes the packet, so we also
// calculate the checksum for the packet and place it in the appropriate
// field.

void init_ping_packet(ICMPHeader* icmp_hdr, int packet_size, int seq_no)
{
    // Set up the packet's fields
    icmp_hdr->type = ICMP_ECHO_REQUEST;
    icmp_hdr->code = 0;
    icmp_hdr->checksum = 0;
    icmp_hdr->id = (USHORT)GetCurrentProcessId();
    icmp_hdr->seq = seq_no;
    icmp_hdr->timestamp = GetTickCount();

    // "You're dead meat now, packet!"
    const unsigned long int deadmeat = 0xDEADBEEF;
    char* datapart = (char*)icmp_hdr + sizeof(ICMPHeader);
    int bytes_left = packet_size - sizeof(ICMPHeader);
    while (bytes_left > 0) {
```

```
        memcpy(datapart, &deadmeat, min(int(sizeof(deadmeat)),
                bytes_left));
        bytes_left -= sizeof(deadmeat);
        datapart += sizeof(deadmeat);
    }

    // Calculate a checksum on the result
    icmp_hdr->checksum = ip_checksum((USHORT*)icmp_hdr, packet_size);
}


////////////////////////////// send_ping /////////////////////////////
// Send an ICMP echo ("ping") packet to host dest by way of sd with
// packet_size bytes.  packet_size is the total size of the ping packet
// to send, including the ICMP header and the payload area; it is not
// checked for sanity, so make sure that it's at least
// sizeof(ICMPHeader) bytes, and that send_buf points to at least
// packet_size bytes.  Returns < 0 for failure.

int send_ping(SOCKET sd, const sockaddr_in& dest, ICMPHeader* send_buf,
        int packet_size)
{
    // Send the ping packet in send_buf as-is
    cout << "Sending " << packet_size << " bytes to " <<
            inet_ntoa(dest.sin_addr) << "..." << flush;
    int bwrote = sendto(sd, (char*)send_buf, packet_size, 0,
            (sockaddr*)&dest, sizeof(dest));
    if (bwrote == SOCKET_ERROR) {
        cerr << "send failed: " << WSAGetLastError() << endl;
        return -1;
    }
    else if (bwrote < packet_size) {
        cout << "sent " << bwrote << " bytes..." << flush;
    }

    return 0;
}


////////////////////////////// recv_ping /////////////////////////////
// Receive a ping reply on sd into recv_buf, and stores address info
// for sender in source.  On failure, returns < 0, 0 otherwise.
//
// Note that recv_buf must be larger than send_buf (passed to send_ping)
// because the incoming packet has the IP header attached.  It can also
// have IP options set, so it is not sufficient to make it
// sizeof(send_buf) + sizeof(IPHeader).  We suggest just making it
// fairly large and not worrying about wasting space.

int recv_ping(SOCKET sd, sockaddr_in& source, IPHeader* recv_buf,
        int packet_size)
{
    // Wait for the ping reply
    int fromlen = sizeof(source);
    int bread = recvfrom(sd, (char*)recv_buf,
            packet_size + sizeof(IPHeader), 0,
            (sockaddr*)&source, &fromlen);
    if (bread == SOCKET_ERROR) {
        cerr << "read failed: ";
        if (WSAGetLastError() == WSAEMSGSIZE) {
            cerr << "buffer too small" << endl;
```

```
        }
        else {
            cerr << "error #" << WSAGetLastError() << endl;
        }
        return -1;
    }

    return 0;
}


//////////////////////////// decode_reply ////////////////////////////
// Decode and output details about an ICMP reply packet.  Returns -1
// on failure, -2 on "try again" and 0 on success.

int decode_reply(IPHeader* reply, int bytes, sockaddr_in* from)
{
    // Skip ahead to the ICMP header within the IP packet
    unsigned short header_len = reply->h_len * 4;
    ICMPHeader* icmphdr = (ICMPHeader*)((char*)reply + header_len);

    // Make sure the reply is sane
    if (bytes < header_len + ICMP_MIN) {
        cerr << "too few bytes from " << inet_ntoa(from->sin_addr) <<
                endl;
        return -1;
    }
    else if (icmphdr->type != ICMP_ECHO_REPLY) {
        if (icmphdr->type != ICMP_TTL_EXPIRE) {
            if (icmphdr->type == ICMP_DEST_UNREACH) {
                cerr << "Destination unreachable" << endl;
            }
            else {
                cerr << "Unknown ICMP packet type " << int(icmphdr->type) <<
                        " received" << endl;
            }
            return -1;
        }
        // If "TTL expired", fall through.  Next test will fail if we
        // try it, so we need a way past it.
    }
    else if (icmphdr->id != (USHORT)GetCurrentProcessId()) {
        // Must be a reply for another pinger running locally, so just
        // ignore it.
        return -2;
    }

    // Figure out how far the packet travelled
    int nHops = int(256 - reply->ttl);
    if (nHops == 192) {
        // TTL came back 64, so ping was probably to a host on the
        // LAN -- call it a single hop.
        nHops = 1;
    }
    else if (nHops == 128) {
        // Probably localhost
        nHops = 0;
    }

    // Okay, we ran the gamut, so the packet must be legal -- dump it
    cout << endl << bytes << " bytes from " <<
```

```
            inet_ntoa(from->sin_addr) << ", icmp_seq " <<
            icmphdr->seq << ", ";
    if (icmphdr->type == ICMP_TTL_EXPIRE) {
        cout << "TTL expired." << endl;
    }
    else {
        cout << nHops << " hop" << (nHops == 1 ? "" : "s");
        cout << ", time: " << (GetTickCount() - icmphdr->timestamp) <<
                " ms." << endl;
    }

    return 0;
}
```

---

rawping.h

```
/**************************************************************************
 rawping.h - Declares the types, constants and prototypes required to
    use the rawping.cpp module.
**************************************************************************/

#define WIN32_LEAN_AND_MEAN
#include <winsock2.h>

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

// Minimum ICMP packet size, in bytes
#define ICMP_MIN 8

#ifdef _MSC_VER
// The following two structures need to be packed tightly, but unlike
// Borland C++, Microsoft C++ does not do this by default.
#pragma pack(1)
#endif

// The IP header
struct IPHeader {
    BYTE h_len:4;           // Length of the header in dwords
    BYTE version:4;         // Version of IP
    BYTE tos;               // Type of service
    USHORT total_len;       // Length of the packet in dwords
    USHORT ident;           // unique identifier
    USHORT flags;           // Flags
    BYTE ttl;               // Time to live
    BYTE proto;             // Protocol number (TCP, UDP etc)
    USHORT checksum;        // IP checksum
    ULONG source_ip;
    ULONG dest_ip;
};

// ICMP header
struct ICMPHeader {
    BYTE type;          // ICMP packet type
    BYTE code;          // Type sub code
```

```
    USHORT checksum;
    USHORT id;
    USHORT seq;
    ULONG timestamp;    // not part of ICMP, but we need it
};

#ifdef _MSC_VER
#pragma pack()
#endif

extern int setup_for_ping(char* host, int ttl, SOCKET& sd,
        sockaddr_in& dest);
extern int send_ping(SOCKET sd, const sockaddr_in& dest,
        ICMPHeader* send_buf, int packet_size);
extern int recv_ping(SOCKET sd, sockaddr_in& source, IPHeader* recv_buf,
        int packet_size);
extern int decode_reply(IPHeader* reply, int bytes, sockaddr_in* from);
extern void init_ping_packet(ICMPHeader* icmp_hdr, int packet_size,
        int seq_no);
```

---

ip_checksum.cpp

```
/**********************************************************************
 ip_checksum.cpp - Calculates IP-style checksums on a block of data.
**********************************************************************/

#define WIN32_LEAN_AND_MEAN
#include <windows.h>

USHORT ip_checksum(USHORT* buffer, int size)
{
    unsigned long cksum = 0;

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

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

    // Return the bitwise complement of the resulting mishmash
    return (USHORT)(~cksum);
}
```

---

ip_checksum.h

```
extern USHORT ip_checksum(USHORT* buffer, int size);
```

Footnotes

1. The traceroute utility ("tracert.exe") works by setting the TTL field to 1, sending a ping, waiting for the reply, setting TTL to 2...and so on. By looking at the addresses returned in the ICMP_TTL_EXPIRE replies, you can "trace" a route through the Internet. Eventually, you'll get an ICMP_ECHO reply, which lets you know when you've completed the route to the host. (Incidentally, many Unix traceroute utilities use UDP instead of ICMP, which if nothing else doesn't require that you use raw sockets.)

2. Finding the next hop on the network can be useful, because it allows you to discover a gateway to another network, such as the Internet. To do this, set the TTL field to 1, send the ping and see who responds with ICMP_TTL_EXPIRE. This isn't reliable, but it can be useful in some situations.

This space intentionally left blank. **:)**

[<< Ping: ICMP.DLL Method] [Passing Sockets Between Processes >>]

Updated Sun Jan 18 2015 04:24 MST [Go to my home page]