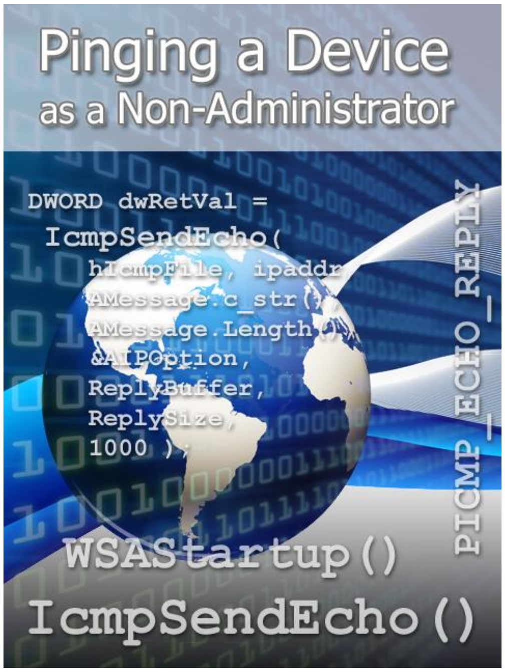
Pinging a Device as a Non-Administrator

By Malcolm Smith

Versions: C++Builder XE-2006, V6-V1



```
DWORD dwRetVal =
  IcmpSendEcho(
    hIcmpFile, ipaddr
    AMessage.c_str(),
    AMessage.Length(),
    &AIPOption,
    ReplyBuffer,
    ReplySize,
    1000 );
```

The Indy sockets library installed with C++Builder includes a component called `TIdIcmpClient` that can be used to ping a remote computer using the Internet Control Message Protocol (ICMP). This implementation utilizes the support of raw sockets made available by WinSock 2.

A raw socket (of type `SOCK_RAW`) allows for direct sending and receiving of network packets by applications, bypassing all encapsulation in the networking software of the operating system. As such, they provide the ability to manipulate the underlying transport so they can be used for malicious purposes that pose a security threat. For this reason, only members of the Administrators group can create raw sockets on Windows 2000 and later.

In this article, I'll be demonstrating an alternative approach using Microsoft APIs that allow you to ping a network device under a non-administrator account.

Echo basics

A ping is the process of sending an echo message to an IP address and reading the reply to verify a connection between TCP/IP hosts. Such a ping can be performed using the `IcmpSendEcho()` family of functions, which is exported by ICMP.DLL on Windows 2000 and IPHLPAPI.DLL on Windows XP and later.

If you are writing code that needs to support Windows 2000, then you need dynamically load IPHLPAPI.DLL and check for the availability of the ICMP functions. If the `GetProcAddress()` function fails to locate the functions then you need to fall back to ICMP.DLL and try again.

Sending an echo

The `IcmpSendEcho()` function sends an IPv4 ICMP echo request to a specified destination IP address and returns any replies received within the timeout specified. `IcmpSendEcho()` is synchronous, so if called from the main thread it will block the UI and message loop. To avoid that, call `IcmpSendEcho()` in a worker thread. In order to keep the accompanying demo application simple to follow, a worker thread is not used. See [2-5] for further information on threading.

If you want to send an asynchronous ping, use the `IcmpSendEcho2()` and `IcmpSendEcho2Ex()` functions. If you want to send IPv6 pings, use the `Icmp6SendEcho2()`function. I will not be covering these functions in this article, but more information can be obtained from MSDN at [1].

To send an IPv4 ping request you need to perform the following steps:

- Initialize WinSock;
- Create an ICMP context handle;
- Prepare and send the echo;
- Close the context handle;
- Cleanup WinSock.

The provided source code only has minimal error handling so be sure to refer to the possible error codes documented on MSDN. As I walk through the code I will be sure to point out what you need to look for.

Figure 1 shows the application at runtime attempting to ping the provided IP address (or domain name) four times, logging the results of each echo response.

The "Ping Now" button's `OnClick` event handler, `btnPingv4Click()`, is implemented like so:

```
void __fastcall TfrmMain::btnPingv4Click(
   TObject *Sender )
 {
   WSADATA AWSAData;
   int AWSAStartupRes = ::WSAStartup(
     MAKEWORD( 1, 1 ), &AWSAData );

   if ( 0 != AWSAStartupRes )
     throw Exception(
       "WSAStartup() failed, "Error: " +
        IntToStr( AWSAStartupRes )
        );
   try
   {
     Memo1->Lines->Clear( );
     Application->ProcessMessages();

     const int cMaxEchoCount = 4;
     for( int i = 0; i < cMaxEchoCount; ++i )
     {
       SendPing( edIPv4Address->Text,
         "< message check >" );

       if( ( cMaxEchoCount - 1 ) != i )
       {
         Log( "" );
         ::Sleep( 1000 );
       }
     }
   }
   __finally
   {
     ::WSACleanup();
   }
 }
```

Although not documented on the IcmpSendEcho() related MSDN pages, you need to initialize WinSock via the WSAStartup() function. I accidentally discovered this while reading [6].

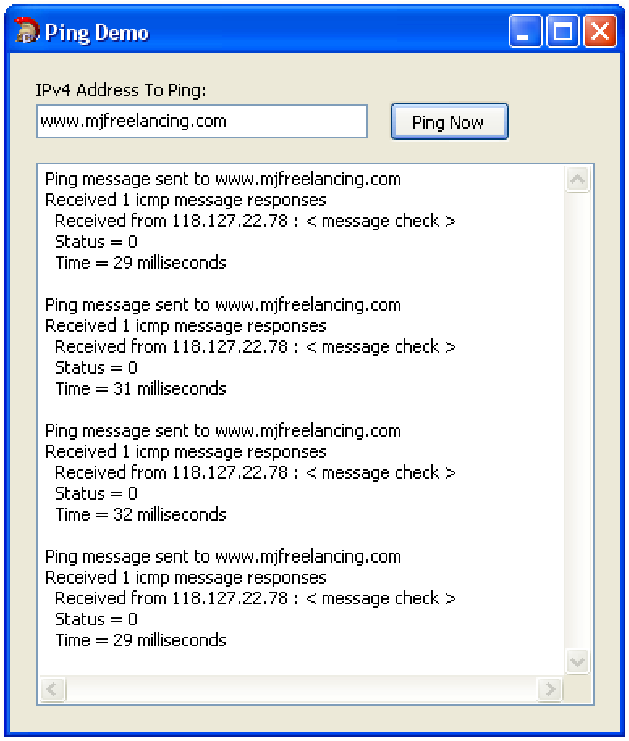
In the above code, WSAStartup() is called like so:

```
WSADATA AWSAData;
int AWSAStartupRes = ::WSAStartup(
  MAKEWORD( 1, 1 ), &AWSAData );
```

WSAStartup() returns zero if successful, otherwise one of the following values (refer to [7] for details):

- WSASYSNOTREADY
- WSAVERNOTSUPPORTED
- WSAEINPROGRESS
- WSAEPROCLIM
- WSAEFAULT

If WinSock is successfully initialized the code next loops four times, calling an internal function called SendPing(), before finally cleaning up WinSock via WSACleanup().



```
Ping message sent to www.mjfreelancing.com
Received 1 icmp message responses
 Received from 118.127.22.78 : < message check >
 Status = 0
 Time = 29 milliseconds

Ping message sent to www.mjfreelancing.com
Received 1 icmp message responses
 Received from 118.127.22.78 : < message check >
 Status = 0
 Time = 31 milliseconds

Ping message sent to www.mjfreelancing.com
Received 1 icmp message responses
 Received from 118.127.22.78 : < message check >
 Status = 0
 Time = 32 milliseconds

Ping message sent to www.mjfreelancing.com
Received 1 icmp message responses
 Received from 118.127.22.78 : < message check >
 Status = 0
 Time = 29 milliseconds
```

Figure 1: *Demo application at runtime pinging the www.mjfreelancing.com domain.*

The SendPing() function is implemented like so:

```
void __fastcall TfrmMain::SendPing(
  const AnsiString &AIPv4Address,
  const AnsiString &AMessage )
{
  if( AMessage.IsEmpty() )
    throw Exception(
      "Cannot echo an empty message" );

  IPAddr ipaddr =
    ::inet_addr( AIPv4Address.c_str() );

  if( INADDR_NONE == ipaddr )
  {
    HOSTENT *hostServer = ::gethostbyname(
      AIPv4Address.c_str() );

    if( NULL != hostServer )
    {
      if( AF_INET == hostServer->h_addrtype )
      {
        ipaddr = *( ( u_long* )
          hostServer->h_addr_list[0] );
      }
    }
  }

  if ( INADDR_NONE == ipaddr )
    throw Exception(
      "Unable to resolve IP Address" );
  HANDLE hIcmpFile = ::IcmpCreateFile( );

  if ( INVALID_HANDLE_VALUE == hIcmpFile )
    RaiseLastOSError( );

  DWORD ReplySize =
    sizeof( ICMP_ECHO_REPLY ) +
    AMessage.Length() + 8;

  std::vector< byte > ABuffer( ReplySize );

  LPVOID ReplyBuffer = &( ABuffer[ 0 ] );

  IP_OPTION_INFORMATION AIPOption;
  ::ZeroMemory( &AIPOption,
    sizeof( AIPOption ) );
  AIPOption.Ttl = 128;
  AIPOption.Flags = IP_FLAG_DF;

  const DWORD cTimeOut = 1000;

  DWORD dwRetVal = ::IcmpSendEcho(
    hIcmpFile, ipaddr, AMessage.c_str(),
    AMessage.Length(), &AIPOption,
    ReplyBuffer, ReplySize,
    cTimeOut );

  if ( 0 != dwRetVal )
  {
    PICMP_ECHO_REPLY pEchoReply =
      reinterpret_cast<
        PICMP_ECHO_REPLY >( ReplyBuffer );

    struct in_addr ReplyAddr;
    ReplyAddr.S_un.S_addr =
      pEchoReply->Address;

    Log( "Ping message sent to " +
      edIPv4Address->Text );

    for( DWORD i = 0; i < dwRetVal; ++i )
    {
      Log( "Received " + String(dwRetVal)
        + " icmp message responses" );
      Log( "  Received from " + String(
        inet_ntoa( ReplyAddr ) ) +
        " : " + AnsiString( ( char* )(
        pEchoReply->Data ),
        pEchoReply->DataSize ) );
      Log( "  Status = " + String(
        pEchoReply->Status ) );
      Log( "  Time = " + String(
        pEchoReply->RoundTripTime ) +
        " milliseconds" );
    }
  }
  else
  {
    Log( "Call to IcmpSendEcho failed" );

    DWORD dwErr = ::GetLastError();
    switch( dwErr )
    {
      case IP_REQ_TIMED_OUT:
        Log( " - Timed Out" );
        break;

      default:
        Log( "Error: " + String( dwErr ) );
        break;
    }
  }

  ::IcmpCloseHandle( hIcmpFile );
}
```

SendPing() takes two parameters. The first is the address of where to send the echo; this can be an IP address (IPv4) or a domain address. I'll explain how this is handled shortly. The second parameter is the message to be sent to the remote device; this message will be returned if the echo is successful.

The IcmpSendEcho() function requires the destination address to be of type IPAddr (defined as ULONG and can be treated as a in_addr struct). SendPing() first attempts to convert the address via inet_addr():

```
IPAddr ipaddr =
  ::inet_addr( AIPv4Address.c_str() );
```

If the address is not a valid IPv4 dotted-decimal address inet_addr() will fail with a result of INADDR_NONE. In this scenario, SendPing() next assumes the address to be a domain and proceeds to resolve the address via gethostbyname().

Once the address has been converted to a IPAddr, SendPing() next creates the required ICMP context handle via IcmpCreateFile(), like so:

```
HANDLE hIcmpFile = ::IcmpCreateFile();
```

IcmpSendEcho() requires an allocated buffer large enough to hold the contents of at least one echo response (of type ICMP_ECHO_REPLY) as well as the original message sent. The buffer must also be large enough to hold eight more bytes (the size of an ICMP error message). In SendPing(), this buffer is created by using a std::vector<byte> as highlighted in green in the above code.

Next, highlighted in blue, the code defines the TTL (time to live) and timeout period (in milliseconds). With everything configured, the echo can finally be sent via IcmpSendEcho(), like so:

```
DWORD dwRetVal =
  ::IcmpSendEcho( hIcmpFile, ipaddr,
    AMessage.c_str(), AMessage.Length(),
    &AIPOption, ReplyBuffer,
    ReplySize, cTimeOut );
```

All of the logging code is highlighted in yellow. If `IcmpSendEcho()` is successful, the return value will be the number of `ICMP_ECHO_REPLY` structures stored in the reply buffer. In my own testing, I only ever seem to get one response, even when I made the reply buffer large enough to contain multiple responses. I would not recommend assuming this will always be the case.

If the call to `IcmpSendEcho()` returns zero, the value of `GetLastError()` will indicate the cause of the problem. There is one small check you need to make though, as seen in the following error handling:

```
DWORD dwErr = ::GetLastError();
switch( dwErr )
{
  case IP_REQ_TIMED_OUT
    // report a “timed out” event
    break;

  default:
    // process error indicated by dwErr
    break;
}
```

It took me a while to work it out (only to eventually discover other developers have experienced the same), but a time-out condition is indicated by error code 11010, which in the API corresponds to `WSA_QOS_ADMISSION_FAILURE` (error due to lack of resources). It just so happens that this error code has the same value as `IP_REQ_TIMED_OUT`.

If the ping is successful, the response data is contained in the `ReplyBuffer` variable. This buffer needs to be cast back to a pointer of type `ICMP_ECHO_REPLY`, which is declared like so:

```
typedef struct icmp_echo_reply {
    IPAddr  Address;
    ULONG   Status;
    ULONG   RoundTripTime;
    USHORT  DataSize;
    USHORT  Reserved;
    PVOID   Data;
    struct ip_option_information Options;
} ICMP_ECHO_REPLY, *PICMP_ECHO_REPLY;
```

In `SendPing()`, the reply buffer is cast to a `PICMP_ECHO_REPLY`. The `Address` member indicates where the response has been delivered from. To convert this to a human-readable string it needs to first be assigned to an `in_addr` type, like so:

```
in_addr ReplyAddr;
ReplyAddr.S_un.S_addr = pEchoReply->Address;
```

The conversion is possible because `in_addr` contains a union (`S_un`) in which `S_addr` is a `ULONG` (which is the same as an `IPAddr`—phew!).

After the reply address has been converted to an `in_addr`, it—along with the `RoundTripTime` and `Data` members—can be converted to a string for logging. Refer to [8] for a detailed description of possible `Status` values.

Conclusion

This article looked at how to perform an IPv4 ping (ICMP echo) using the `IcmpSendEcho()` function under Windows 2000 and above when the user account does not have administrator privileges.

I mentioned that `IcmpSendEcho()` is exported by ICMP.DLL in Windows 2000 and IPHLPAPI.DLL in Windows XP and above. For this reason, I discussed how you should dynamically load IPHLPAPI.DLL and test for the presence of the ICMP functions (unlike the demo, which has been statically linked to IPHLPAPI.LIB).

The remainder of the article looked at how to interpret the results, including time-out conditions.

Contact Malcolm at **msmith@bcbjournal.com**.

References

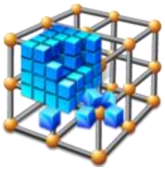
1. The `IcmpSendEcho()` related functions are detailed at http://tinyurl.com/4r7t5dn.
2. M. Smith, “Serious Threading, Part I,” *C++Builder Dev. Journal*, **10** (8), 2006.
3. M. Smith, “Serious Threading, Part II,” *C++Builder Dev. Journal*, **10** (9), 2006.
4. M. Smith, “Serious Threading, Part III,” *C++Builder Dev. Journal*, **10** (11), 2006.
5. M. Smith, “Serious Threading, Part IV,” *C++Builder Dev. Journal*, **11** (2), 2007.
6. http://support.microsoft.com/kb/170591
7. `WSAStartup()` is documented at http://tinyurl.com/4ztd9y2
8. The `ICMP_ECHO_REPLY` status values are defined at http://tinyurl.com/4joslln.

Develop software faster without sacrificing reliability using MJFAF and MJFVCL

— Exclusive to C++Builder —

New Version 2 of these products not only extends on the reliability of Version 1, but also offers many new features and has been greatly simplified to only a handful of runtime packages making them even easier to use.

Full Unicode support for C++Builder 2009 to XE is now available.



MJFAF (*MJ Freelancing Application Framework*) for C++Builder 5 through XE is a comprehensive set of runtime packages designed to increase development throughput and product stability through the use of well known design patterns and compile-time enforced type safety.

http://www.mjfreelancing.com/mjfaf.htm

MJFVCL (*MJ Freelancing VCL*) for C++Builder is a component suite that complements MJFAF and the standard VCL. The components utilize many of the MJFAF runtime packages, thereby ensuring that the proven record of reliability is carried through to your applications.

http://www.mjfreelancing.com/mjfvcl.htm

Purchase MJFAF and receive a FREE copy of MJFVCL.

Just $99 for 1-year upgrades/support
$299 for lifetime upgrades/support

Subscribers of the BCB Journal will receive the following benefits:

- Priority e-mail support
- 6-month extension of your BCB Journal subscription

For more information visit
http://www.mjfreelancing.com

MJ Freelancing

Key areas include...

- Advanced debugging aids
- Applications licensing
- Dynamic object lifetime management
- Advanced threading
- Adding multiple thread access safety to objects
- Implementing object and thread pools
- Manage shared memory
- Implement TCP/IP communication using Indy
- Perform data hashing (MD4/MD5/SHA1/SHA2)
- Perform data encryption (AES—Rijndael)
- Perform data compression (BZIP2)
- Access the Windows Background Intelligent Transfer Service
- Accessing the Recycle Bin
- Easily perform bit masking calculations
- Obtain detailed information about various versions of Windows
- Create (thread safe) atomic data types
- Dynamically load DLLs with added type safety
- Implement advanced streams
- Access environment variables
- COM utilities such as the Global Interface Table
- Generate HTML using an object based framework
- Read/Write XML files using an object based framework
- Read/Write INI files using an object based framework
- Structured configuration files with import/export to INI/XML files and the Registry. Export to HTML is also available.
- MDI form management
- Access Runtime Type Information of VCL objects
- Create secondary desktops
- Implement Singleton based objects
- Create deletion functors for use with STL containers
- Access the Windows System Restore Points
- Access the Windows Firewall service
- Use the Windows Management Interface
- Info Balloons
- Tray icon
- Advanced file search operations
- Thread safe logging
- Inter-process communications using named pipes
- Data storage in PE header
- Advanced data storage in STL collections offering policy based thread safety and copy semantics
- and much, much more...