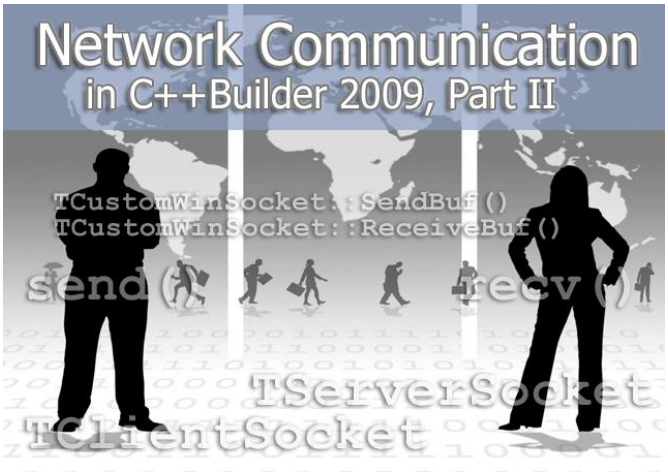
Network Communication in C++Builder 2009, Part II



By George Tokas and Remy Lebeau

Versions: C++Builder 2009

Last month's article described specific changes that need to be applied in C++ Builder 2009 code to work with network communications that are based on ideas in previous articles. That article triggered a difference of opinion between George Tokas and Remy Lebeau in the Journal's discussion forums.

Who are these people?

George is a contributing editor of the Journal. As you may have read in past articles, he has expertise in graphics, multimedia, and network communications.

Remy is a member of TeamB [1] and is on the development team for the open-source Indy communications library [2]. He has extensive experience in network communications.

As you can understand, both have significant experience, and whatever both state has a point. Difference of opinions is not something bad. Most of the time, it leads to evolution.

George's socket history

In 1999, George started experimenting with socket communications. At that time, there was the "Chat" example shipped with C++Builder, which was located under the "Examples" subdirectory of C++ Builder's installation. Binary-based communication was not mentioned in the docs nor were there any examples. (Around the same time, early versions of Indy appeared, stable but a bit out of his league at that time.) Also, there were no TCP socket components that supported the IPv6 protocol.

C++ Builder 6 and earlier version had `TClientSocket` and `TServerSocket` components installed on the "Internet" tab of the Component Palette. These components were later deprecated in favor of new `TTcpClient` and `TTcpServer` components in Borland's CLX cross-platform framework. However, these latter components were not well-suited for general-purpose use, and have since been replaced with Indy. The legacy `TClientSocket` and `TServerSocket` components are not installed by default anymore, but can be installed manually if needed, as described in previous articles.

George's choice at that time (and until now) has been to use the `TClientSocket` and `TServerSocket` components. He had to decide between the available socket configurations: to use blocking or non-blocking sockets. Based on his experience as an electronic engineer—20 years of field experience before 1999—and the physical design of the network end of his communications (RJ45 style "serial" exchange) he saw no reason not to use a non-blocking configuration. In addition, the size of his data packets at the time was less than 100 bytes each, which tends to work well with non-blocking sockets.

Using a non-blocking configuration over a LAN worked perfectly for George without any corruption of packets, and in his experience has worked with 100% success all these years. The only problem he found was that when PCs go into power-save mode, data packets might be lost. Because of that, he added "keep alive" functions on both sides of his communications, which exchange strings at random periods between 2 to 4 minutes of idle activity.

As the years passed, he added security to his

communications, and the strings grew larger and larger. In one project, he had the following format exchanged:

1. String: 3 to 20 bytes of garbage data + Start string sequence + the actual String + End string sequence + 5 to 20 bytes of garbage data.
2. This string was encrypted using DEC and one algorithm and at this time, the string has just the encrypted data.
3. To that, a hash string sequence separator was added, along with the hash sequence string itself (usually 40 bytes).

This process was employed in multiple applications using other encryption and hashing algorithms, and the string about to be transmitted was larger than 1 KB. Even in that "extreme" condition, there were no problems, not even a single packet drop. The time the server side of his communications needed to process such a scenario and respond back (decrypt, check the hash, decrypt again, etc.) was less than 1 ms using a humble 1.8 Ghz PC.

Expanding the network, he used his scenario on an ISDN 64 Kb connection again without any problems. The communication was expanded to the Internet without any problem as well. Then ADSL came, and for testing reasons, he stretched an early 1MB/256KB connection to its limit using a LAN that had many machines using 99.99% of the bandwidth to the Internet, with one machine acting as a server application. Again, no loss of packets occurred, even when that machine used part of the network bandwidth for reasons other than the server application.

In all those years, George considered the `TClientSocket` and `TServerSocket` components as just tools. Only when he started using the user class approach in his server-side code did he have to look inside the source code for those components. There, he found out that all communications are actually binary even though he was using strings.

Remy's perspective on the VCL socket components

When used properly, the `TClientSocket` and `TServerSocket` components work well to transmit and receive string data and binary data alike. Remy has been using them almost as long as George has, in both blocking and non-blocking configurations, with much success.

However, by studying the VCL source code, Remy has learned that you must take into account how these components work internally. More importantly, you have to understand how sockets handle data packets, or else your socket I/O will have hidden bugs in it. You may not see problems occur in your daily communications (like George), but the potential for data loss and data corruption will still be present nonetheless. It is only a matter of time before those bugs will cause problems in your communications, and you won't know why if you do not take the necessary precautions beforehand.

The `SendText()` and `ReceiveText()` methods of the `TCustomWinSocket` class transmit and receive string values over the socket. They are implemented in SCKTCOMP.PAS, and are particularly troublesome in all versions of C++Builder. The rest of this article will explain reasons why.

The SendText() method

The `SendText()` method is implemented in C++Builder 2007 and earlier, and in the initial release of C++Builder 2009, as follows:

```
function TCustomWinSocket.SendText(
  const s: string): Integer;
begin
  Result := SendBuf(Pointer(S)^,
    Length(S));
end;
```

It is important to point out that the return value indicates the number of bytes that were sent by the socket, not the number of characters. In C++ Builder 2007 and earlier, Delphi's `string` data type mapped to `AnsiString`, which uses single-byte characters, so the result was effectively the same as the number of characters. In C++ Builder 2009, however, the `string` data type now maps to a new `UnicodeString` type [3]-[6]. This means the above implementation is broken in early releases of C++ Builder 2009!

Fortunately, this bug was partially fixed in a later update, as follows:

```
function TCustomWinSocket.SendText(
  const s: AnsiString): Integer;
begin
  Result := SendBuf(Pointer(S)^,
    Length(S) * SizeOf(AnsiChar));
end;
```

However, this fix introduces a new bug in C++Builder 2009 (that has not been fixed yet at the time of this writing). AnsiString is now codepage-aware, so it is possible to store Unicode data into an AnsiString using any codepage that the OS supports. Assigning any string (Ansi or Unicode) that uses one codepage to any other string (Ansi or Unicode) that uses a different codepage will automatically perform the necessary data conversion for you. However, this causes a problem for SendText(). An AnsiString variable that does not have an explicit codepage associated with it at compile-time (such as the input parameter of SendText()) will use the OS default codepage. This means that any string passed to SendText() will now perform a data conversion to the OS default codepage before transmission. There is no guarantee that the receiver is using the same default codepage, and the receiver has no way of knowing which codepage was actually used in the transmitted data. So, that is one potential area of data loss, as the characters actually transmitted may be different form the characters that are passed to SendText().

To make SendText() behave correctly in C++ Builder 2009, while maintaining backward compatibility with earlier versions, CodeGear should have declared SendText() to accept a RawByteString (which is another new string type in C++ Builder 2009) instead of an AnsiString, like so:

```
function TCustomWinSocket.SendText(
  const s: RawByteString): Integer;
begin
  Result := SendBuf(Pointer(S)^,
    Length(S) * SizeOf(AnsiChar));
end;
```

Without going into details, just know that assigning any AnsiString value, regardless of its codepage, to a RawByteString will not perform a data conversion. Using RawByteString would have allowed SendText() to transmit any kind of Ansi data correctly.

The SendBuf() method

SendText() has another more subtle bug in it, in all versions of C++ Builder, because it misuses the SendBuf() method. SendBuf() is implemented in all C++Builder versions as follows:

```
function TCustomWinSocket.SendBuf(
  var Buf; Count: Integer): Integer;
var
  ErrorCode: Integer;
begin
  Lock;
  try
    Result := 0;
    if not FConnected then Exit;
    Result := send(FSocket, Buf, Count, 0);
    if Result = SOCKET_ERROR then
    begin
      ErrorCode := WSAGetLastError;
      if (ErrorCode <> WSAEWOULDBLOCK) then
      begin
        Error(Self, eeSend, ErrorCode);
        Disconnect(FSocket);
        if ErrorCode <> 0 then
          raise ESocketError.CreateResFmt(
            @sWindowsSocketError,
            [SysErrorMessage(ErrorCode),
            ErrorCode, 'send']);
      end;
    end;
  finally
    Unlock;
  end;
end;
```

The WinSock API send() function is being called only one time to send the entire data block. As long as the data that are being sent fits within a single TCP/IP packet, everything is fine. But if the data do not fit, the send() function only sends what it can, leaving the remaining portion of the data unsent. This is a very important fact in socket programming, regardless of whether a blocking or non-blocking configuration is used. The return value of the send() function indicates the actual number of bytes that were sent. It is the caller's responsibility to check that value and call send() again if there are bytes still waiting to be sent. This applies to all uses of send(), not just for string data.

This behavior of the send() function inside of SendBuf() means that SendText() has another potential area for data loss: It does not call SendBuf() more than one time if the input string is too long for the socket to transmit in a single TCP/IP packet. This is especially important when send() returns SOCKET_ERROR and WSAGetLastError() then returns WSAEWOULDBLOCK, which indicates that the input string was not sent at all.

Your code needs to look at the return value of SendText () and act accordingly. If the return value is –1 (and no OnError event was fired), then re-send the same string again as-is. If the return value is 0, either the socket has been disconnected, or the string was empty. If the return value is greater than 0 but

less than the length of the string, then ignore the characters that were successfully sent and call SendText() again with a new string containing just the remaining characters. Otherwise, the entire string has been sent.

The above approach works fine in C++Builder 2007 and earlier, but unfortunately can be problematic in C++ Builder 2009 because of the Unicode bugs mentioned above.

The ReceiveText() method

Similar problems exist in the ReceiveText() method, which is implemented in all versions of C++Builder as follows:

```
function TCustomWinSocket.ReceiveText:
  string;
begin
  SetLength(Result,
    ReceiveBuf(Pointer(nil)^, -1));
  SetLength(Result,
    ReceiveBuf(Pointer(Result)^,
      Length(Result)));
end;
```

ReceiveText() does not work correctly in C++Builder 2009 because of the new UnicodeString mapping of Delphi's string data type. Unlike SendText(), ReceiveText() was not updated to use AnsiString, so it ends up allocating memory for a UTF-16 Unicode string, but then fills it half-way (at most) with single-byte characters. The result is garbage.

The ReceiveBuf() method

A more subtle bug, in all versions of C++Builder, is again a misuse of the WinSock API, this time with the ReceiveBuf() method, which is implemented in C++Builder 2006 and later as follows:

```
function TCustomWinSocket.ReceiveBuf(
  var Buf; Count: Integer): Integer;
var
  ErrorCode, iCount: Integer;
begin
  Lock;
  try
    Result := 0;
    if (Count = -1) and FConnected then
      ioctlsocket(FSocket, FIONREAD,
        Longint(Result))
    else begin
      if not FConnected then Exit;
      if ioctlsocket(FSocket, FIONREAD,
        iCount) = 0 then
      begin
        if (iCount > 0) and (iCount < Count)
        then
          Count := iCount;
      end;

      Result :=
        recv(FSocket, Buf, Count, 0);
      if Result = SOCKET_ERROR then
      begin
        ErrorCode := WSAGetLastError;
        if ErrorCode <> WSAEWOULDBLOCK then
        begin
          Error(Self, eeReceive,
            ErrorCode);
          Disconnect(FSocket);
          if ErrorCode <> 0 then
          raise ESocketError.CreateResFmt(
             @sWindowsSocketError,
             [SysErrorMessage(ErrorCode),
             ErrorCode, 'recv']);
        end;
      end;
    end;
  finally
    Unlock;
  end;
end;
```

ReceiveBuf() is implemented slightly differently in earlier versions of C++Builder 2006, but the differences are not enough to change the outcome of the bug.

In both cases, the problem is similar to that of SendText(). ReceiveBuf() is not called enough times to receive all of the data for a given string if it could not fit in a single TCP/IP packet. ReceiveText() has ReceiveBuf() call the WinSock API ioctlsocket() function to find out how many bytes are currently pending in the socket's incoming data buffer, and then call recv() to actually read them. If the network connection is slow, or if ReceiveText() is simply called before all of the data have been received, ReceiveBuf() only reads the bytes that are currently available on the socket, if any, and returns the number of bytes that were actually received. This is another potential area of data loss, as ReceiveText() does not wait for all of the data to arrive.

Unlike when using SendText(), your code cannot look at the return value of ReceiveText() in order to act accordingly. It will have to call ReceiveBuf() directly instead. If the return value is –1, or if the return value is 0 and the Count parameter is –1, then no data are yet available but the connection is still alive. If the return value is 0 and the Count parameter is not –1, either the socket has been disconnected, or the Count parameter was 0. If the return value is greater than 0,

then that many bytes were read. If you are expecting more bytes, then you have to call `ReceiveBuf()` again with an adjusted `Count` parameter.

A potential area of data corruption exists when receiving data. If the sending party sends multiple data packets close together, `ReceiveText()` might end up receiving them together (due to the way sockets cache outbound data into the most efficient data packets they can send), thus returning a single string that actually contains multiple (possibly partial) data packets in it. Your code may end up ignoring the extra data, thus not having it available for later processing. Worse, it may end up processing the entire string as a whole, causing incorrect results, e.g., if hashes or encryption are invoked.

Working around the bugs

Because a socket can transmit and receive large data blocks in smaller packets, and can receive multiple data blocks together, `SendText()` and `ReceiveText()` are both places where data loss and/or corruption can occur, in all versions of C++Builder, because neither of them really handle transmissions correctly (Unicode bugs aside). A better approach would be to simply avoid `SendText()` and `ReceiveText()` altogether and use `SendBuf()` and `ReceiveBuf()` directly for everything—which George discussed in Part I of this series. This technique will work in all versions of C++Builder, for both blocking and non-blocking configurations, and for string and binary data alike. (For blocking configurations, you actually have to use the `TWinSocketStream` class instead, but that is a separate detail.) Both methods accept raw memory pointers as parameters, so you can send whatever you want, and receive whatever you want, without worry of any data conversions occurring that you do not perform yourself.

Here's the evolution: When sending data, keep track of how many bytes `send()` actually accepts each time, and keep calling `send()` until you reach the end of your outgoing data. When receiving data, keep track of how many bytes `recv()` actually reads each time, and keep calling `recv()` until you reach the end of the expected data. This way, you will be in a better position to handle errors and partial transmissions more accurately.

Yet, how do you know how much incoming data to expect, so you know how much reading to do? Well, there are three ways to handle that.

The best way is to send the data size before sending the actual data. The receiver can then read that size first, allocate a memory block of that size, and keep reading bytes into it until it fills up. However, that is not always possible in all protocols (like most Internet text-based protocols such as SMTP/POP3, NNTP, FTP, etc.).

Another option is to include some kind of unique delimiter at the end of each data block you send, where the delimiter does not appear in the data itself (if it does, it would have to be escaped). The receiver can then read all incoming bytes into an intermediate buffer, and when the delimiter is encountered then process and remove the front bytes from that buffer up to, and including, the delimiter, leaving behind any remaining bytes that have already been received for subsequent packet(s) for later read operations to consume.

The final option is a combination of the two: Use a fixed-length or delimited header that contains the data size should. This header information would precede the actual data (HTTP and many binary protocols do this).

Conclusions

Both authors have a point, based on each one's experience. Differences of opinions lead to productive discussion, which— like the discussion presented here— leads to evolution. This evolution is based on cooperation and mutual understanding.

From George Tokas, a special thanks to Remy Lebeau for this insightful discussion.

Contact George at **gtokas@tokas-bros.eu**
Contact Remy at **remy@lebeausoftware.org**

References

1. http://www.teamb.com
2. http://www.indyproject.org
3. http://edn.embarcadero.com/article/38437
4. http://edn.embarcadero.com/article/38498
5. http://edn.embarcadero.com/article/38693
6. http://edn.embarcadero.com/article/38980