Reverse-engineering MSN Messenger's Video Conversation Format

Ramiro Polla

March 2006

1 Introduction

This article is an attempt to document the communications protocol used by Video Conversations in MSN Messenger.

The main goal of reverse-engineering this protocol is to have alternative means to decode it, and make it easier for open-source softwares to also implement it and be able to communicate with the official MSN Messenger client.

The data analysed for this article refers to the actual video and audio data, and does not document all the steps necessary to get there (within a Switchboard Session in MSN Messenger). This kind of information is documented by some other documents on the Internet, and if I have time, I might document them myself.

Also, there are a few data packets that I do not understand yet, but they're unnecessary to decode the audio and video packets.

In this article, the data will be explained in three formats. UDP Packets refer to the information the way it was received from the Internet. Video Conversation Packets refer to a single packet as defined in the Single Packet section. Many single packets can be used to make up an Specific Packet, such as a Video Packet.

2 Transmission

The network layer protocol used in the data I gathered is the Internet Protocol. It uses a high numbered port to send, and low numbered port to receive.

Both parties will necessarily be sending and receiving video, or else MSN Messenger would instead use the Webcam feature, which uses a different kind of communications protocol. Information on this other protocol can be found on the same site you've downloaded this article.

The transport layer protocol used for transferring the Video Conversation data between the sender and the receiver is the User Datagram Protocol. The choice of using UDP over Transfer Control Protocol is probably due to the fact that the latter is not very reliable for real-time video and audio transfers.

Since UDP does not guarantee a reliable and in-order delivery of packets, MSN Messenger implements a way to re-transmit missing or corrupt information, which will be seen latter on.

Some conversations start with 4 or 8 packets that I have not been able to identify the meaning of. The first byte is either 0x00 or 0x01. I just ignore them for the moment. They don't seem to influence anything in the video or audio data.

More than one Video Conversation packet may be sent in the same UDP packet. The packets will not be divided in less than a Single Packet though.

3 Single Packet Structure

The packet structure is defined by a 10-byte header followed by the payload. The size of the payload is defined by the *size* value in the header. Different kinds of packet (determined by the *code* value) have different ways to deal with the header and data.

The packet structure is defined by the following C type structure:

```
struct vc_header
{
    u_int8_t    code;
    u_int16_t   retransmission:5;
    u_int16_t   size:11;
    u_int8_t    frame_chunk:6;
    u_int8_t    nkeyframe:2;
    u_int32_t   timestamp;
    u_int8_t    frame_number;
    u_int8_t    frame_chunks;
};
```

The naming of the variables is based on the way Video Packets use the header. Other kinds of data will have a different use for some of the variables, but the data sizes are the same.

This structure must be carefully implemented. It uses the *little-endian* byte order. Since it is optimized for space, the data is *1 byte packed*. Pay special attention to the values that do not use up full bytes.

To achieve 1 byte packed data using Microsoft's C Compiler, use a *#pragma pack(1)* statement right before the structure declaration. After the declaration, remember to reset the byte alignment, using a *#pragma pack( )* statement.

To achieve the same using GNU's C Compiler, use __attribute__ ((packed)) right after the } symbol closing the structure definition.

The correct meaning of the byte that contains nkeyframe and frame_chunk is not known yet. Having 6 bits for frame chunk seems overkill since no frames will be so huge. Also, the !keyframe value is probably a 1-bit flag.

Following is an explanation on each variable inside the structure, along with their usual values.

code

This byte will determine what kind of data the payload is. Possible values are:

- *0x44 - 01000100*: Syn/Ack.
- *0x48 - 01001000*: Authorization.
- *0x4a - 01001010*: Audio.
- *0x62 - 01100010*: Video.
- *0x66 - 01100110*: Connection.

retransmission

This 5-bit value (0-31) is a counter used in re-transmitted packets. If the receiver notices the packet has been corrupted in the transmission, it will request a re-transmission. This value is incremented each time a same packet is sent. If a re-transmitted packet is received, the former packet must be discarded. *Re-transmitted packets are always contain the same data, except for this counter.*

size

This 11-bit value (0-2048) determines the size of the payload that follows the header.

frame_chunk

This 6-bit value (0-4096) contains the chunk number in a chunked Specific Packet. It start at 0, like in acessing arrays in C.

nkeyframe

This flag (used here as 2-bit, but is probably 1-bit only) is set when a Video Packet is not a keyframe. In other kinds of data, it means nothing, and is set to zero.

timestamp

This double word value is the timestamp (obtained with GetTickCount()) in Video Packets. Since Video Packets do not follow a framerate, it is used to keep the correct time.

In audio packets, it is a counter to determine which packet (starting from 0) it is. Since audio packets do have a constant playback time, it is not necessary to send an exact timestamp.

frame_number

This byte determines the frame number this current packet belongs to. Using *frame_chunk* and *frame_chunks*, a frame packet can be reconstructed.

It is a counter, and being an unsigned byte, it goes back from 255 to 0. During a bad transfer, you may be receiving data belonging to many frames out-of-order.

frame_chunks

This number, consistent for all the packets with the same *frame_number*, means the number of packets a Specific Packet (normally video) was split up into.

4 Joining Single Packets

This section only refers to Video Packets (*code* set to 0x62). For this section, we're only dealing with Video Packets, and all other Single Packets are ignored.

They contain a *frame_number* value to identify which frame the packet belongs to. That value is an unsigned byte, so it will wrap to 0 once it reaches 255. That is not a problem, since you will never need to keep so many frames waiting around to complete transmission. Once you reach, let's say, frame number 50, the previous frame that used the same *frame_number* will be long gone.

In a working Video Conversation, for any given *frame_number*, there should be *frame_chunks* Single Packets, with the *frame_chunk* value starting in 0 and ending in (*frame_chunks* - 1), just like an array in C.

In the case of a bad connection, retransmission of packets with the exact same *frame_number* and *frame_chunk* will be sent, and they should be used to replace the previous Single Packet with the same *frame_number* and *frame_chunk*. In the case you receive 2 or more Single Packets with the same *frame_number* and *frame_chunk*, the ones with smaller retransmission value should be discarded.

After you have all the chunks of a frame, you can reconstruct the full frame. Just concatenate the payload of the Single Packet starting from *frame_number* 0 up to (*frame_chunks* - 1).

Implementation Example

If you're making a list of the whole stream to create an index, it is a good idea to first make a list of all the Single Packets in the correct order, discarding corrup packets.

it is best to use a double-linked list, so that you can browse back and forth through the stream. Also, while populating the list, make sure you read through the starting by the last entry, and not the first. Is it natural that the packets you're receiving belong to the latest frames, so browsing from the end gives better performance.

C++ has the STL to make the job of handling lists much easier, but here I'll show a double-linked list algorithm example to implement the ordering of the Single Packets received in a double-linked list.

```
// A packet should contain the header and the data.
struct vc_packet
{
    vc_header   header;
    u_int8_t   *data;
};

// Start list
first_packet = first_vc_packet( );
last_packet = first_packet;

// Populate it
while( new_packet = next_packet( ) )
{
    for( tmp_packet = last_packet ;
         tmp_packet exists ;
         tmp_packet = tmp_packet->previous )
    {
        // Compare new_packet with tmp_packet
        if both frame_numbers are the same
        {
            if both frame_parts are the same
            {
                if new_packet's retransmission is greater
                    replace tmp_packet with new_packet
                else
                    discard new_packet
            }
            else if new_packet's frame_part is greater
                put new_packet after tmp_packet
        }
        else if new_packet's frame_part is greater
            put new_packet after tmp_packet
    }

    if no replacement ocurred (rare)
    {
        put new_packet before first_packet,
        make it the new first_packet
    }
}
```

```
// All packets are ordered, starting from first_packet
```

After ordering the list, you could go through it removing frames that do not contain all the chunks.

You could then dump the frame data into another struct, which would contain the full frame information in one single place. Chunk and re-transmission information are no longer necessary. Here's an example to extract the data for one given *frame_number*. It uses C's memory function prototypes.

```
struct vc_frame_header
{
    u_int8_t    code;
    u_int16_t   size;
    u_int8_t    nkeyframe;
    u_int32_t   timestamp;
    u_int8_t    frame_number;
};

struct vc_frame
{
    vc_frame_header header;
    u_int8_t *data;
};

// Start frame
frame.header.code = first_packet->header.code;
frame.header.size = 0; // Will be incremented for every Single Packet
frame.header.nkeyframe = first_packet->header.nkeyframe;
frame.header.timestamp = first_packet->header.timestamp;
frame.header.frame_number = first_packet->header.frame_number;
frame.data = NULL;

tmp_vc_packet = first_packet;

while( tmp_vc_packet->header.frame_number == frame.header.frame_number )
{
    u_int16_t newsize = frame.header.size + tmp_vc_packet->header.size;
    frame.data = realloc( frame.data, newsize );
    memcpy( frame.data + frame.header.size, tmp_vc_packet->header.size );
    frame.header.size = newsize;
    tmp_vc_packet = tmp_vc_packet->next;
}
```

5 Specific Packets

A Specific Packet is just a Single Packet for most codes, except for the Video Packet, which is chunked. For the Video Packet, it is a full frame as explained in last section.

From now on, "Packet" refers to a "Specific Packet".

Each kind of Packet (determined by the header's code) must be handled differently. A sub-section will be given to every code.

5.1 Code 0x44 - Syn/Ack

This Packet is *sent* for every Video Packet *received* as an acknowledgement, and vice-versa.

If you have sent a Video Packet and not received an acknowledgement in a given timeout (not studied by me, but probably just a few seconds), you must send a re-transmission, and expect an acknowledgement for the re-transmission.

The values set in the header are the *size*, which is set to the 3, the *frame_chunk*, which is set to 1, and the *frame_chunks*, which is also set to 1. The rest is set to zero.

The Payload is made up of 3 bytes. They correspond to the Single Packet to acknowledge, and are made up of this C struct:

```
struct syn_data
{
    u_int8_t    frame_number;
    u_int8_t    frame_chunk;
    u_int8_t    retransmission;
};
```

```
Example Header:
44 60 00 01 00 00 00 00 00 01
Example Payload:
0a 02 01
```

Which is an acknowledgement for the following Video Packet (explained in a while):

```
Video Packet Header:
62 41 4d 02 9a 66 b3 02 0a 04
```

5.2 Code 0x48 - Authorization

This Packet is transmitted a couple of times before the actual Video and Audio data are sent. All its header values are zero, except for *code*. Since *size* is also zero, it contains no payload.

```
Example Header:
48 00 00 00 00 00 00 00 00 00
```

5.3 Code 0x4a - Audio

I still don't know how Audio is codec'd in Video Conversations. What is known is that the codec used is "MSN Messenger Audio Codec", the same as the Voice Clip feature uses. The file that codec belongs to is *sirenacm.dll*, but I don't understand how the encoded data is passed to it yet. Any help is appreciated.

I kind of know what the Header means already. The values set are *size*, *frame_chunk* and *frame_chunks*, which are set to 1, and *timestamp*, but it will not contain a timestamp. It will contain a counter for the Audio Frame. That counter is incremented for every Audio Frame, and its timestamp can be calculated knowing the time of the first audio frame, since all audio frames take the same time.

Audio Frames are not re-transmitted in case one fails.

```
Example Header:
4a 00 0a 01 2d 01 00 00 00 01
```

5.4 Code 0x62 - Video

Video Packets are codec'd using the "WMV3" codec. The file that codec belongs to is *wmv9vcm.dll*, located in MSN Messenger's folder.

The decoding must start with a keyframe. Those frames will have their header's *nkeyframe* flag unset. So, if the "not keyframe" flag is not set, (!!keyframe == keyframe), you have a keyframe, and it can be fed to the codec for decompressing. Subsequent frames can then be fed to the codec for decompressing one by one. To go back one frame, you must find the previous keyframe, and decode from it to the point you want.

In this article, I will give an example on how to decode a video frame using Win32 API. First, the library must be loaded, then the codec must be opened. After that, frames can be decompressed, and after you're done, you must close the codec, and possibly free the library.

I'm assuming the input and output are 320x240 pixels, but I think using this codec you might be able to define other output sizes. There is an extra 6 bytes of data (which I name *SomeData*) that I don't know the exact meaning of. But using these same 6 bytes always works. The output is a 320x240 pixels RGB (3 bytes) Bitmap, so the memory used by it is 320 x 240 x 3 = 230400 = 0x38400 bytes.

```
// Include needed headers
#include <windows.h>
#include <vfw.h>

// Define data that will be needed
struct WMV3BitmapInfoHeaderPlusSomeData
{
    BITMAPINFOHEADER BIH;
    char SomeData[6];
```

```
};

char SomeData[] = { 0x0f, 0xf1, 0x80, 0x01, 0x40, 0x0f };
WMV3BitmapInfoHeaderPlusSomeData WMV3;
BITMAPINFOHEADER BITM;
HMODULE wmv9vcm;
FARPROC DriverProc;
HIC hIC;
char *output;

// Initialize data
memset( &WMV3, 0x00, sizeof(WMV3) );
WMV3.BIH.biSize = sizeof(BITMAPINFOHEADER) + 6;
WMV3.BIH.biWidth = 320;
WMV3.BIH.biHeight = 240;
WMV3.BIH.biPlanes = 1;
WMV3.BIH.biBitCount = 0x18;
WMV3.BIH.biCompression = FOURCC_WMV3;
WMV3.BIH.biSizeImage = 0x038400;
memcpy( &WMV3.SomeData, &SomeData, sizeof(SomeData) );

memset( &BITM, 0x00, sizeof(BITM) );
BITM.biSize = sizeof(BITMAPINFOHEADER);
BITM.biWidth = 320;
BITM.biHeight = 240;
BITM.biPlanes = 1;
BITM.biBitCount = 0x18;
BITM.biSizeImage = 0x038400;

// Start codec
wmv9vcm = LoadLibrary( "wmv9vcm.dll" );
DriverProc = GetProcAddress( wmv9vcm, "DriverProc" );
ICInstall( ICTYPE_VIDEO, FOURCC_WMV3, (LPARAM) DriverProc, NULL, ICINSTALL_FUNCTION );
hIC = ICOpen( ICTYPE_VIDEO, FOURCC_WMV3, ICMODE_DECOMPRESS );
ICDecompressBegin( hIC, &WMV3, &BITM );

// Assuming "frame" contains a header and the data:
output = malloc( 0x038400 );
WMV3.biSizeImage = frame->header.size;
ICDecompress( hIC, 0, &WMV3, frame->data, &BITM, output );

// "output" now contains the decoded frame
// Do the above steps as many times as you wish...

//Clean up
ICDecompressEnd( hIC );
```

```
ICClose( hIC );
ICRemove( ICTYPE_VIDEO, FOURCC_WMV3, 0 );
FreeLibrary( wmv9vcm );
free( output );
```

5.5 Code 0x66 - Connection

This Packet is transmitted a couple of times before the actual Video and Audio data are sent. It is the same string used in the Webcam feature, and depends on the information exchanged to create the connection between the sender and the receiver (in MSN Messenger's Switchboard Session).

```
Example Header:
66 60 04 01 00 00 00 00 01 01
Example Payload:
"recipientid=100&sessionid=1347\r\n\r\n"
```

6 More Info

The studies explained in this article were made so that the program "MSN Webcam Recorder" could support Video Conversations recording and playback. Except for having an exceptionally ugly and messy source code, it works in decoding both Webcam streams and Video Conversation streams. For the moment, it does not support Audio in Video Conversation. It can be found, along with complete source code, at http://ml20rc.msnfanatic.com