A Case for Calculating UDP Throughput Using Wireshark and uC/TCP-IP

One way to gauge the performance of a TCP-IP stack or TCP-IP based application is to calculate its throughput; that is, how many bits per second can be processed by the device from the physical layer to the application layer. One of the most popular tools to perform throughput tests is IPerf, which has several implementations out there but is in its purist form a cross-platform command line application that acts as a client or server to transmit or receive a data payload to the remote host. The tool essentially allows the user to specify how much data to send, over which transport, at what time interval and for how long; yielding a detailed report at the end of the test (Figure 1).

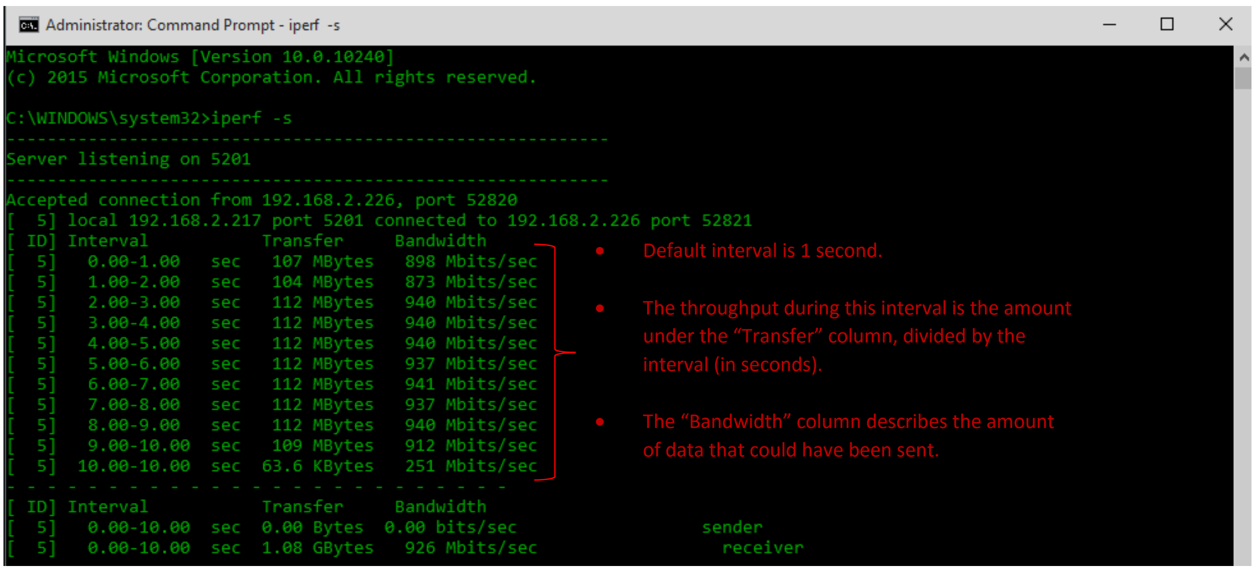

Figure 1: Typical IPerf results screen.

The problem with calculating throughput this way is that embedded implementations of IPerf might use other protocols such as TELNET or Serial to output the intermediary results at each interval which might introduce undesired overhead that (depending on the microprocessor's performance) may negatively impact the throughput figures.

For this reason, a case could be made to isolate these externalities from the device under test and conduct throughput tests manually and independently, using a combination of Wireshark, WinSock2 or BSD socket programming, and the embedded stack. Wireshark provides a capture summary (by clicking on Statistics -> Capture File Properties on the menu bar) that quickly lists the throughput of a TCP stream and transferred UDP datagrams. However, unlike TCP, the UDP protocol itself has no way to acknowledge the received data back to the sender. If the PC were to act as a client and our embedded device as a server, we can never know how many of those captured frames made it through the TCPIP stack and reached the application layer by using the Wireshark method.

To solve this, we can conceive a uC/OS-III application task (like the AppUDP_ServerTask() shown in Figure 3) that instantiates a UDP server (App_UDP_Server() in Figure 4) whose only role is to listen for an incoming connection on a specified port (UDP_SERVER_PORT/20002) and consume whatever data has been received in chunks determined by RX_BUF_SIZE.

```c
/*
*********************************************************************************************************
*                                             INCLUDE FILES
*********************************************************************************************************
*/

#include  "app_cfg.h"
#include  "app_tcpip.h"
#include  <lib_str.h>
#include  <os.h>
#include  <Source/net_sock.h>
#include  <Source/net_app.h>


/*
*********************************************************************************************************
*                                             LOCAL DEFINES
*********************************************************************************************************
*/

#define  RX_BUF_SIZE                       1472u
#define  TX_BUF_SIZE                       1472u
#define  UDP_CLIENT_PORT                  20001u
#define  UDP_SERVER_PORT                  20002u

#define  IP_ADDR                          "192.168.2.20"
#define  SUBNET_MASK_ADDR                 "255.255.255.0"
#define  DFLT_GATEWAY_ADDR                "192.168.2.1"

#define  LOCAL_TEST_UDP_SERVER_IP_ADDR   "192.168.2.42"
#define  LOCAL_TEST_TCP_SERVER_IP_ADDR   "192.168.2.42"


/*
*********************************************************************************************************
*                                              DATA TYPES
*********************************************************************************************************
*/


/*
*********************************************************************************************************
*                                         LOCAL GLOBAL VARIABLES
*********************************************************************************************************
*/

static   CPU_CHAR    ServerData[RX_BUF_SIZE];
static   CPU_STK     AppTaskUDPServerStk[APP_CFG_TASK_TCP_SERVER_STK_SIZE];
static   OS_TCB      AppTaskDatagramServerTCB;
volatile CPU_INT32U  Hit_Rate_Ctr = 0u;


/*
*********************************************************************************************************
*********************************************************************************************************
*                                       GLOBAL FUNCTION PROTOTYPES
*********************************************************************************************************
*********************************************************************************************************
*/

void  App_UDP_Server    (void);
void  AppUDP_ServerTask (void  *p_arg);
```

Figure 2: Definitions and declarations for UDP server instance.

```c
void  AppUDP_ServerTask (void *p_arg)
{
    OS_ERR  err;


    (void)p_arg;

    while (DEF_ON) {
        App_UDP_Server();
        OSTimeDlyHMSM( 0u, 0u, 0u, 1u,
                       OS_OPT_TIME_HMSM_STRICT,
                      &err);
    }
}
```

Figure 3: UDP server task definition.

```c
void  App_UDP_Server (void)
{
            NET_SOCK_ID          sock;
            NET_IPv4_ADDR        server_ipv4_addr;
            NET_SOCK_ADDR_IPv4   server_sock_addr_ip;
            NET_SOCK_ADDR_IPv4   client_sock_addr_ip;
            NET_SOCK_ADDR_LEN    client_sock_addr_ip_size;
            NET_SOCK_RTN_CODE    rx_size;
            CPU_CHAR             rx_buf[RX_BUF_SIZE];
            CPU_CHAR            *p_ip_addr;
            CPU_BOOLEAN          fault_err;
            NET_ERR              err;


    p_ip_addr = "0.0.0.0";                                  /* Set server's IPv4 address to 'this host'.            */

    NetASCII_Str_to_IP( p_ip_addr,                          /* Convert string representation to a 32-bit integer.   */
                       &server_ipv4_addr,
                        NET_IPv4_ADDR_SIZE,
                       &err);
                                                            /* ----------------- OPEN IPV4 SOCKET ----------------- */
    sock = NetSock_Open((NET_SOCK_PROTOCOL_FAMILY) NET_SOCK_ADDR_FAMILY_IP_V4,
                                                   NET_SOCK_TYPE_DATAGRAM,
                                                   NET_SOCK_PROTOCOL_UDP,
                                                  &err);
    if (err != NET_SOCK_ERR_NONE) {
        return;
    }
                                                            /* ------------ CONFIGURE SOCKET'S ADDRESS ------------ */
    NetApp_SetSockAddr((NET_SOCK_ADDR *)&server_sock_addr_ip,
                                         NET_SOCK_ADDR_FAMILY_IP_V4,
                                         UDP_SERVER_PORT,
                          (CPU_INT08U *)&server_ipv4_addr,
                                         NET_IPv4_ADDR_SIZE,
                                        &err);
    switch (err) {
        case NET_APP_ERR_NONE:
            break;


        case NET_APP_ERR_FAULT:
        case NET_APP_ERR_NONE_AVAIL:
        case NET_APP_ERR_INVALID_ARG:
        default:
            NetSock_Close(sock, &err);
            return;
    }
                                                            /* ------------------- BIND SOCKET -------------------- */
    NetSock_Bind(                   sock,
                 (NET_SOCK_ADDR *)&server_sock_addr_ip,
                                   NET_SOCK_ADDR_SIZE,
                                  &err);
    if (err != NET_SOCK_ERR_NONE) {
        NetSock_Close(sock, &err);
        return;
    }

    fault_err = DEF_NO;
    Mem_Clr(&rx_buf, sizeof(rx_buf));
    do {
                                                            /* ----- WAIT UNTIL RECEIVING DATA FROM A CLIENT ------ */
        client_sock_addr_ip_size = sizeof(client_sock_addr_ip);
        rx_size = NetSock_RxDataFrom(                  sock,
                                                       ServerData,
                                                       RX_BUF_SIZE,
                                                       NET_SOCK_FLAG_NONE,
                                     (NET_SOCK_ADDR *)&client_sock_addr_ip,
                                                      &client_sock_addr_ip_size,
                                                       DEF_NULL,
                                                       DEF_NULL,
                                                       DEF_NULL,
                                                      &err);
        switch (err) {
            case NET_SOCK_ERR_NONE:                         /* Do nothing. This is not an echo server.              */
                 Hit_Rate_Ctr++;
                 break;


            case NET_SOCK_ERR_RX_Q_EMPTY:
            case NET_ERR_FAULT_LOCK_ACQUIRE:
                 break;


            default:
                 fault_err = DEF_YES;
                 break;
        }

    } while (fault_err == DEF_NO);
    (void)rx_size;
                                                            /* ------------- FATAL FAULT SOCKET ERROR ------------- */
    NetSock_Close(sock, &err);                              /* This function should be reached only when a fatal ...*/
                                                            /* fault error has occurred.                            */
}
```

Figure 4: UDP server instance.

On the client side, one could simply have a Winsock or BSD UDP client that fires frames down to the microsecond range (or as fast as the hardware will allow) in order to stress-test the device.

```c
#include  <winsock2.h>
#include  <stdio.h>
#include  <windows.h>
#include  <string.h>

void usleep(__int64 usec);

short  RemotePort;                                                      /* Communication Port for remote host.                  */

#define  ECHO_SERVER_LOCAL_PORT           20002
#define  BUFLEN                           1472                          /* Max length of buffer.                                */
#define  TIMEOUT_uS                       100                           /* Interval between sent datagrams (in microseconds).   */
#define  REMOTE_HOST                      "192.168.2.20"                /* IP address of remote host.                           */


/*
*****************************************************************************************************************************
*                                                       main()
*
* Description: Application entry point. This function is responsible for setting up the communication WinSock socket and
*              transmitting the data buffer to REMOTE_HOST every TIMEOUT_uS milliseconds.
*****************************************************************************************************************************
*/

int main(void)
{
    struct    sockaddr_in server_sock_info;
    struct    sockaddr_in client_sock_info;
    int       s;
    int       slen = sizeof(server_sock_info);
    char      broadcastEnable;
    int       ret;
    int       payload_size;
    char      buf[BUFLEN];
    char*     message_ptr;
    WSADATA   winsock_data;
    long      packet_cnt = 0;
    char*     str = "This is client data\n";


    RemotePort   =  ECHO_SERVER_LOCAL_PORT;
    message_ptr  = &buf[0u];
    payload_size =  BUFLEN;
                                                                        /* Fill out payload.                                    */
    for (int i = 0; i < sizeof(buf) / strlen(str); i++) {
        strcpy(&buf[i * strlen(str)], str);
    }

    printf("\nInitializing Winsock...");
    if (WSAStartup(MAKEWORD(2,2),&winsock_data) != 0)
    {
        printf("Failed. Error Code : %d",WSAGetLastError());
        exit(EXIT_FAILURE);
    }
    printf("Initialized.\n");
                                                                        /* Create socket.                                       */
    if ((s = socket(AF_INET, SOCK_DGRAM, 0)) == SOCKET_ERROR) {
        printf("socket() failed with error code : %d" , WSAGetLastError());
        exit(EXIT_FAILURE);
    }

    memset((char *) &server_sock_info, 0, sizeof(server_sock_info));

    server_sock_info.sin_family           = AF_INET;
    server_sock_info.sin_port             = htons(RemotePort);
    server_sock_info.sin_addr.S_un.S_addr = (unsigned long)inet_addr(REMOTE_HOST);

                                                                        /* Send out a million datagrams.                        */
    for (int i = 0; i < 1000000; i++) {
        if (sendto(s, message_ptr, payload_size , 0 , (struct sockaddr *) &server_sock_info, slen) == SOCKET_ERROR) {
            printf("sendto() failed with error code : %d" , WSAGetLastError());
            exit(EXIT_FAILURE);
        }
        usleep(TIMEOUT_uS);
    }

    closesocket(s);
    WSACleanup();

    return 0;
}


void usleep(__int64 usec)
{
    HANDLE        timer;
    LARGE_INTEGER li;

    li.QuadPart = -(10 * usec);
    timer       = CreateWaitableTimer(NULL, TRUE, NULL);
    SetWaitableTimer(timer, &li, 0, NULL, NULL, 0);
    WaitForSingleObject(timer, INFINITE);
    CloseHandle(timer);
}
```

Figure 5: Winsock-based UDP client.

Since the scope of this blog only covers Winsock, we need to install Cygwin or MinGW so that we can run the gcc compiler and build the executable using the following command in the Windows terminal or Cygwin:

```
gcc udp_c.c -o udp_client.exe -std=c99 -lwsock32
```

and then executing the `udp_client.exe` to run the program. If you notice in line 12 of Figure 5, there is a preprocessor #define (TIMEOUT_uS) that controls how far apart the datagrams are sent out (set to 100 uS) but as stated before, this number can deviate from it due to hardware constraints.

Before `udp_client.exe` is running, we first need to run our UDP server application in the embedded target, and then start a capture in Wireshark calculate the throughput. It's necessary to filter the capture with `ip.addr==192.168.2.20 && !icmp`, replacing 192.168.2.20 with the IP address of the embedded target and clicking on the Start Capture button.

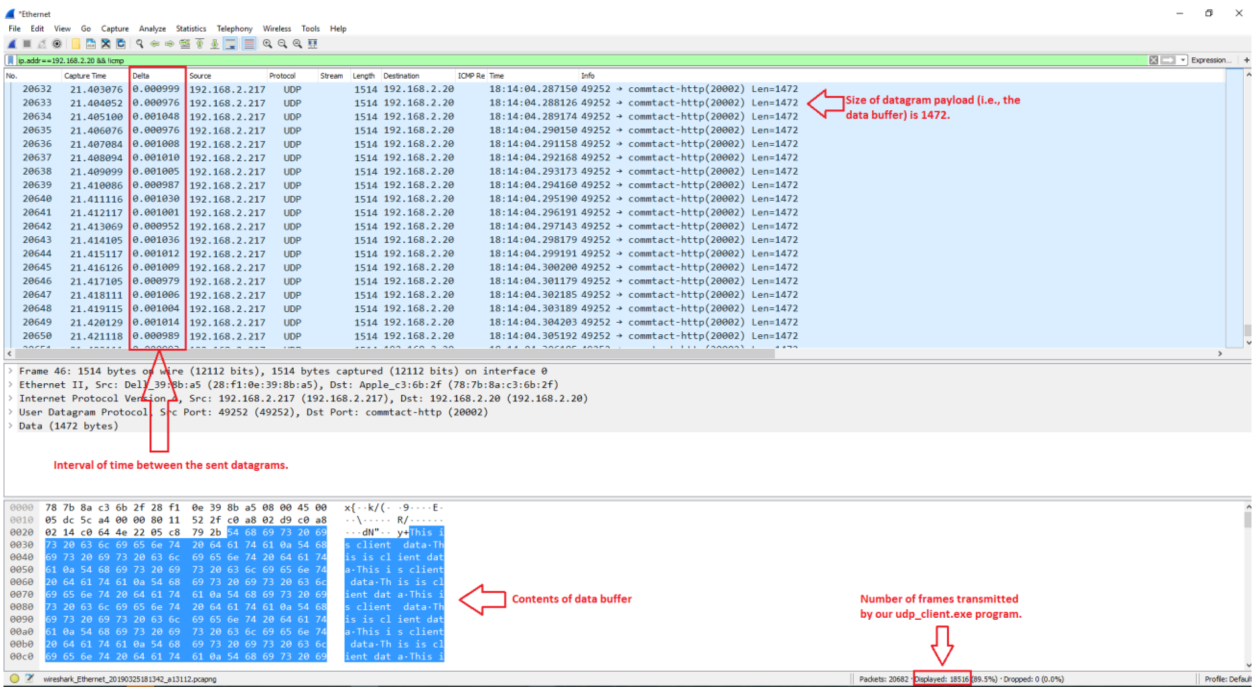

Figure 6: Wireshark capture of UDP datagrams sent out by udp_client.exe

Once udp_client.exe is finished sending datagrams and Wireshark does not show any more them incoming, stop the capture by clicking the button and pause the debug session on the embedded target. If you notice in Figure 4, whenever the call to NetSock_RxDataFrom() returns without error we increment a global variable named `Hit_Rate_Ctr`, which is a simple counter that tallies up how many of the sent frames actually made it to the application layer and were not dropped. We can finally calculate the throughput of this capture in Mbps by plugging all the information we've obtain until this point into the following formula:

```
UDP Rx Throughput (Mbps) =
Hit_Rate_Ctr * [nbr_of_frames * payload_bytes * 8 bits] / [dur_in_sec * 2^20]
```

where `nbr_of_frames` is how many frames were captured with the applied filter and transmitted by our udp_client.exe program (See Figure 6), `payload_bytes` is the length of the payload carried by each datagram (or RX_BUF_SIZE in bytes), and the `dur_in_sec` is how much time has elapsed since the first frame was captured, and NOT how much time has elapsed since the *beginning* of the capture. To make this value easier to calculate a time reference can be added by right-clicking on the first captured frame and clicking on “Set/Unset Time Reference” in the context menu.