Oct 24, 2021

RELIABLE DATA PROTOCOL

1. INTRODUCTION

In this assignment you are going to implement a reliable data protocol in the GINI router (i.e., gRouter). The gRouter is a tiny user-level router that is fully implemented in C. It is about 20000 lines of C code. It includes the TCP/IP stack from LwIP (lightweight IP stack, which is used in many embedded devices). The LwIP provides a near full featured UDP protocol layer for the gRouter. As you know UDP does not provide any reliability. If you send packets from a node to another node and the network is congested in the middle (e.g., a denial of service is taking place), we can have information loss. That is information sent from node A to node B would not reach as intended. With TCP, you would still have packet loss, but there will be retransmissions to recover from the loss. All information sent from node A would appear at node B (the TCP protocol layer could have done many retransmissions to make this happen). The TCP approach could take lot of time (due to retransmissions), but it gets all the information through the lossy network. The UDP, on the other hand, does not do any retransmissions and as a result the information carried in the packets that are being discarded will be lost.

The objective of this assignment is to design and implement a RDP that does what TCP is doing. Note that we are not trying to make RDP equivalent to TCP only similar in the primary objective. While the TCP is a reliable byte-stream protocol, RDP adds reliability to datagrams (e.g., message packets). You will keep track of the datagrams that are sent from the sender to the receiver and expect an acknowledgement from the receiver. If the acknowledgement is not received, we send out a retransmission and repeat this until we hit success.

2. A SIMPLE SCENARIO

Let’s consider the following network topology. We have two routers in the network. Log into the routers. At the router CLI, type help gnc. You will see the manul page for the netcat that is implemented in gRouter as shown in the figure below.

```
GINI-Router_1 $ help gnc
gnc(1)                          gRouter Commands                          gnc(1)

NAME
       gnc - gRouter's nc (netcat) to create TCP and UDP connections and listens

SYNOPSIS
       gnc [-u] <destination> <port>

       gnc [-u] -l <port>

DESCRIPTION
       Create  a  TCP or UDP connection with a remote host. After the connection is estab-
       lished it will behave like a two-way chat, similar to the real nc command.

       Use the -u switch to use UDP instead of the default option of TCP.

       Use the -l switch to specify that gnc should  listen  for  an  incoming  connection
       rather than initiate a connection to a remote host.

       Use  the  <destination>  and  <port> arguments to create a connection with a remote
       host.

AUTHORS
       Written  by  Archit  Agnihotri.  Send  comments  and   feedback   at   archit.agni-
       hotri@mail.mcgill.ca.
```

You can run the gnc (gRouter's netcat) in UDP mode or TCP (default). We are interested in the UDP mode. You can run one instance as the server end and another as client. You can have a text exchange between two instances as shown in the following two figures.

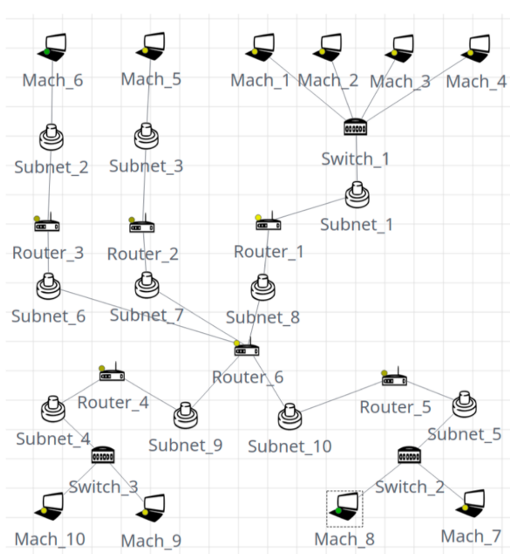
```
GINI-Router_2 $ gnc -u -l 5000
dfdf
dsfsdfdsf
dsfsdfsdfdsf
dsfdsfdsfsdfsdf
dsfdsfsdfdsfsdfsdf
dfsdfsdfdsfds
dfdsfdsfdsf
dfdsff
dfdsfsdfsdf
dfsdfdsfdsfdsfdsfds

sdfdsfds
dfdsfdfdf

dfdsfdsf
```

```
GINI                                15 August 2019
GINI-Router_1 $ gnc -u 192.168.0.129 5000
dfdf
dsfsdfdsf
dsfsdfsdfdsf
dsfdsfdsfsdfsdf
dsfdsfsdfdsfsdfsdf
dfsdfsdfdsfds
dfdsfdsfdsf
dfdsff
dfdsfsdfsdf
dfsdfdsfdsfdsfdsfds
^C
GINI-Router_1 $
GINI-Router_1 $
GINI-Router_1 $
GINI-Router_1 $ gnc -u 192.168.0.129 5000
sdfdsfds
dfdsfdfdf
^C
```

While this session works. It is implemented using UDP. You can launch denial of service attacks and get some examples where the text at the sending side does not match the one at the receiving side. You need to use a larger topology.



The above figure shows the topology you used for the denial-of-service experiments. A smaller (pruned) version of this topology would be sufficient. For instance, Switch_1 need not have four machines. With the sending texts and received texts not matching, you have a problem that needs a solution. Now we want to introduce retransmission into the underlying implementing (i.e., the UDP implementation) such that the texts at both sides match.

3. UDP UNDER THE HOOD

Let's first understand how the **gnc** session shown above is working. The **gnc** is implemented in **cli.c.** In lines 762-771 and 801-810, the cli.c writes the data you type into the terminal through a UDP send to the remote side.

```
761             gncTerm = false;
762             while (!gncTerm) {
763                 fgets(payload, sizeof(payload), stdin);
764
765                 // create pbuf and call udp_send()
766                 struct pbuf *p = pbuf_alloc(PBUF_TRANSPORT, strlen(payload), PBUF_RAM);
767                 p->payload = payload;
768                 err_t e1 = udp_send(pcb, p);
769                 if (e1 != ERR_OK)
770                     printf("udp send error: %d\n", e1);
771             }
```

You will notice that the message is read from the terminal in Line 763 into payload. You make a protocol buffer data structure with that data in Line 766. All networking stacks (including the one in Linux) use a protocol buffer data structure to hold the data that needs communication. This data structure has a linked list for data storage and is capable of handling data that has been collected incrementally (through an interactive session like gnc). The udp_send() in Line 768 has two parameters. One is the protocol buffer that has the payload and the other is the protocol control buffer (PCB) that has the definition of the connection. The PCB itself for the UDP connection was created in the following section of cli.c.

```
751
752             // create and initialze pcb to listen to UDP connections at the specified port
753             struct udp_pcb * pcb = udp_new();
754             udp_recv(pcb, udp_recv_callback, pcb);
755             uchar any[4] = {0,0,0,0};
756             udp_bind(pcb, any, port);
```

In Line 753, a new PCB is constructed, and Line 754 binds a callback handler to process the received datagrams. We bind the UDP connection to the appropriate port in Line 756. The callback handler used in cli.c is shown in the following code snippet.

```
597 /*
598  * callback function for UDP packets received
599  */
600 void udp_recv_callback(void *arg, struct udp_pcb *pcb, struct pbuf *p, uchar *addr, uint16_t port)
{
601     printf("%s", (char*)p->payload);
602     uchar ipaddr_network_order[4];
603     gHtonl(ipaddr_network_order, addr);
604     udp_connect(arg, ipaddr_network_order, port);
605 }
```

You can see something interesting in the above code snippet. In Line 603, the addr is translated to network-byte order from host-byte order. This is important in low level networking. This comes from the fact we have two types byte orders: big endian and little endian. Therefore, we need to translate everything to network-byte order before sending them (i.e., any data that is more than a byte long). Because IPv4 addresses are 4 bytes we are using the gHtonl() function that translates long (4 bytes). Now that we have seen how cli.c is using the UDP code. Let's look at the UDP code to understand what is going on there.

One of the functions you will see in udp.c is the following that computes the UDP checksum. You don't need to modify this function, however, it is educational to know what is going on there. In case, you are hit with a bug, you might want to know the processing that is taking place in the UDP layer!

```
 *
 * @return The UDP checksum for the specified UDP packet.
 */
uint16_t udp_checksum(ip_packet_t *ip_packet)
{
        // Derive UDP packet from IP packet and reset checksum
        udp_packet_type *udp_packet = (udp_packet_type *)
                (ip_packet + (ip_packet->ip_hdr_len * 4));
        udp_packet->checksum = 0;

        // Create UDP pseudo-header
        udp_pseudo_header_type pseudo_header;
        COPY_IP(&pseudo_header.ip_src, ip_packet->ip_src);
        COPY_IP(&pseudo_header.ip_dst, ip_packet->ip_dst);
        pseudo_header.reserved = 0;
        pseudo_header.ip_prot = ip_packet->ip_prot;
        pseudo_header.udp_length = udp_packet->length;

        // Calculate sums of pseudo-header and UDP packet (same as taking one's
        // complement of checksum) and store in temporary buffer
        uint16_t buf[3];
        buf[0] = ~checksum((uint8_t *) &pseudo_header,
                                    sizeof(udp_pseudo_header_type) / 2);
        buf[1] = ~checksum((uint8_t *) &udp_packet, ntohs(udp_packet->length) / 2);
        // If UDP packet length is odd, store zero-padded last byte in temporary
        // buffer as well
        if (udp_packet->length % 2 != 0)
        {
                uint8_t *temp = (uint8_t *) (udp_packet +
                        ntohs(udp_packet->length) - 1);
                buf[2] = *temp << 8;
        }
        else
        {
                buf[2] = 0;
        }

        // Find checksum of the sums (plus the last byte, if applicable)
        return checksum((uint8_t *) &buf, 3);
}
```

The checksum function is straightforward. It extracts the UDP packet from the IP payload (after the header is the payload). It resets the existing checksum and then computes the new checksum. To compute the new checksum, a UDP pseudo header is constructed and the checksum() function is called over the resulting pseudo header.

Now, let's get into more important functions. When a packet comes into the router, it is processed by the local IP stack. If the IP stack figures out that the packet is meant for the local machine (that is it is directly addressed for this router or is a broadcast packet) and the packet type is UDP, it will deliver the packet to the UDP layer by calling the udp_input() function that is partly shown below. You will notice that there is gpacket_t data type (for the second argument) in addition to the protocol buffer data structure. The gpacket_t is the way gRouter handles packets, which might change in future versions.

```
163  */
164 void
165 udp_input(struct pbuf *p, gpacket_t *in_pkt, uchar *netmask, uchar *network)
166 {
167   struct udp_hdr *udphdr;
168   struct udp_pcb *pcb, *prev;
169   struct udp_pcb *uncon_pcb;
170   ip_packet_t *iphdr;
171   uint16_t src, dest;
172   uint8_t local_match;
173   uint8_t broadcast = 0;
174
175   PERF_START;
176
177   iphdr = (ip_packet_t *)p->payload;
178
179   /* Check minimum length (IP header + UDP header)
180    * and move payload pointer to UDP header */
181   if (p->tot_len < (IPH_HL(iphdr) * 4 + UDP_HLEN) || pbuf_header(p, -(s16_t)(IPH_HL(iphdr) * 4))) {
182     /* drop short packets */
183     LWIP_DEBUGF(UDP_DEBUG,
184                 ("udp_input: short UDP datagram (%"U16_F" bytes) discarded\n", p->tot_len));
185     pbuf_free(p);
186     goto end;
187   }
188
189   udphdr = (struct udp_hdr *)p->payload;
190
191   /* is broadcast packet ? */
192   //bradcast = ip_addr_isbroadcast(in_pkt->frame.nxth_ip_addr, inp);
193   broadcast = IPProcessBcastPacket(in_pkt); // this is gini's version (not currently implemented)
194
195   LWIP_DEBUGF(UDP_DEBUG, ("udp_input: received datagram of length %"U16_F"\n", p->tot_len));
196
197   /* convert src and dest ports to host byte order */
198   src = ntohs(udphdr->src);
199   dest = ntohs(udphdr->dest);
200
201   udp_debug_print(udphdr);
```

You notice that the first argument of the udp_input() is a protocol buffer and the second argument is a gpacket_t. Both arguments are holding the incoming packet (i.e., the UDP packet) in different formats. In Line 198 and 199, we extract the source and destination ports, respectively. The source port the port at the sending side of the UDP session. The destination is the current machine (gRouter). You will notice that the port numbers are being converted from network-byte order to host-byte order. This is necessary for all information that is being retrieved from the network.

The udp_input() function calls the callback that was setup in cli.c. You can see in Line 307 the callback is triggered with the correct packet information. This way the application (in this case the gnc) will get the the UDP payload.

```
290
291   /* Check checksum if this is a match or if it was directed at us. */
292   if (pcb != NULL || ip_addr_cmp(network, iphdr->ip_dst)) {
293     LWIP_DEBUGF(UDP_DEBUG | LWIP_DBG_TRACE, ("udp_input: calculating checksum\n"));
294     {
295     }
296     if(pbuf_header(p, -UDP_HLEN)) {
297       /* Can we cope with this failing? Just assert for now */
298       LWIP_ASSERT("pbuf_header failed\n", 0);
299       pbuf_free(p);
300       goto end;
301     }
302     if (pcb != NULL) {
303       //snmp_inc_udpindatagrams();
304       /* callback */
305       if (pcb->recv != NULL) {
306         /* now the recv function is responsible for freeing p */
307         pcb->recv(pcb->recv_arg, pcb, p, iphdr->ip_src, src);
308       } else {
309         /* no recv function registered? then we have to free the pbuf! */
310         pbuf_free(p);
311         goto end;
312       }
313     } else {
314       LWIP_DEBUGF(UDP_DEBUG | LWIP_DBG_TRACE, ("udp_input: not for us.\n"));
315       pbuf_free(p);
316     }
317   } else {
318     pbuf_free(p);
319   }
320 end:
```

The udp_send() is the other major function. You can see it is calling the udp_sendto() and that is calling another function. Eventually, the UDP packet is written to the IP layer and the packet gets into the network from there. So, this description should give you some idea of how the UDP is working. You are highly encouraged to trace the function calls from the send and receive sides of UDP and understand the packet paths.

4. SUGGESTED APPROACH

Instead of gnc calling UDP send and receive (through the callback), we want it to call RDP send and receive. The RDP send and receive would use UDP send and receive but add reliability to it as illustrated in the lectures (textbook). In this assignment you are asked to use the stop-and-wait (S&W) algorithm (this simplest reliable data communication algorithm!). You need to implement the following concepts: timeouts, acknowledgements, checksum. For checksum, you can just reuse the UDP checksum. Implementing timeouts need you to set the start of the timer and then keep decrementing it. The LwIP has timeout implementations, but you are required to rollout your own (simple) implementation of timeout. You should be able to start, reset the timer. Also, when the timer expires the callback will be called. In your case, the callback will be resending the datagram because the timeout indicates that the datagram has been lost. Implementing acknowledgements is the other task. There is no way of setting a flag in a UDP packet. For this there is a trick you can do. You can assume certain bits of the port number field are going to remain unused (in your application!) and repurpose those bits for flags. You just need one bit, but it would not hurt to have few more bits allocated for flags.

Once you have your RDP implementation, modify the cli.c so that gnc uses your implementation. You need to redo the original experiment and demonstrate that there is no message loss despite packet losses.

5. ADDITIONAL NOTES

In the source directory `~/gini5/backend/src/grouter`, you will see a `udp.c` file. You will see all C source files for the gRouter in that directory. The header files that correspond to the source files can be found in the `~/gini5/backend/include` directory. When you make the changes run the following commands to create a new version of the gRouter.

`scons`

`scons install`

After the new version of the gRouter is installed, restart the topology and you should be using the new version.