IPv6 Networking

IPV6 is the cutting edge of the Internet Protocol (IP). IPv6 was developed by IETF to deal with the long-anticipated problem of IPv4 address exhaustion. IPv6 uses 128 bit address and thereby allow approximately $3.4 * 10^{38}$ addresses. Contiki supports IP networking through the uIP TCP/IP stack or through IP routing using the RPL protocol. RPL networks are directed (oriented) towards a root that acts as border router. This border router connects the mesh network to external networks. 6LowPAN is an acronym for IPv6 over low-power wireless personal area networks. It can be regarded as an analogy to IPv6 for resource constrained systems like wireless sensor network. The 6LowPAN defines encapsulation and header compression that allow IPv6 packets to be sent and received over IEEE 802.15.4 based networks. Tunneling is done in order to translate the 6LowPAN data frames into IPv6 data or vice versa.

Task 1: COAP IMPLEMENTATION

Constrained Application Protocol (CoAP) is a web transfer protocol and designed to interface with HTTP especially for the resource constrained devices. The first task of this lab was to create a routing protocol (RPL) for which a border router was added. Several motes were added. These motes run the sky-websense program in them. The screenshot of the network window is shown in figure 1.

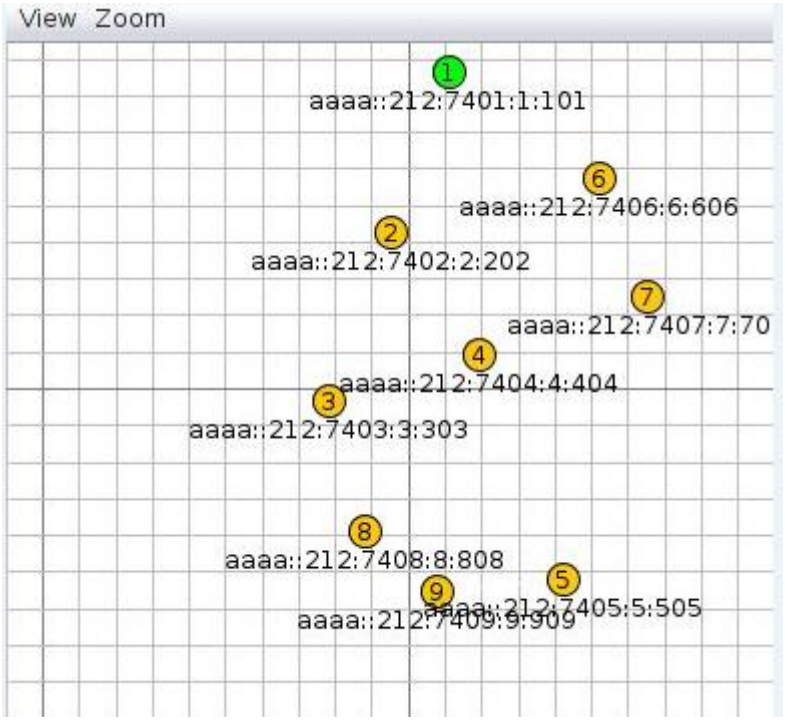

Figure 1: Network window screenshot

Figure 1 shows the network window screenshot from the Cooja simulation. Node 1 runs the border router while other nodes run the sky websense program. This ensures the multihop communication. The border router and all the sensor nodes were pinged to observer their respnonse time values and time to live values (TTL). Figure 2 and Figure 3 shows the ping results.

```
user@instant-contiki:~/contiki-2.7/tools$ ping6 aaaa::212:7401:1:101
PING aaaa::212:7401:1:101(aaaa::212:7401:1:101) 56 data bytes
64 bytes from aaaa::212:7401:1:101: icmp_seq=1 ttl=64 time=0.077 ms
64 bytes from aaaa::212:7401:1:101: icmp_seq=2 ttl=64 time=0.034 ms
64 bytes from aaaa::212:7401:1:101: icmp_seq=3 ttl=64 time=0.055 ms
64 bytes from aaaa::212:7401:1:101: icmp_seq=4 ttl=64 time=0.030 ms
64 bytes from aaaa::212:7401:1:101: icmp_seq=5 ttl=64 time=0.036 ms
64 bytes from aaaa::212:7401:1:101: icmp_seq=6 ttl=64 time=0.037 ms
64 bytes from aaaa::212:7401:1:101: icmp_seq=7 ttl=64 time=0.033 ms
64 bytes from aaaa::212:7401:1:101: icmp_seq=8 ttl=64 time=0.037 ms
64 bytes from aaaa::212:7401:1:101: icmp_seq=9 ttl=64 time=0.032 ms
64 bytes from aaaa::212:7401:1:101: icmp_seq=10 ttl=64 time=0.037 ms
64 bytes from aaaa::212:7401:1:101: icmp_seq=11 ttl=64 time=0.033 ms
64 bytes from aaaa::212:7401:1:101: icmp_seq=12 ttl=64 time=0.036 ms
64 bytes from aaaa::212:7401:1:101: icmp_seq=13 ttl=64 time=0.031 ms
64 bytes from aaaa::212:7401:1:101: icmp_seq=14 ttl=64 time=0.039 ms
```

Figure 2: Pinging results for border-router

```
user@instant-contiki:~/contiki-2.7/tools$ ping6 aaaa::212:7406:6:606
PING aaaa::212:7406:6:606(aaaa::212:7406:6:606) 56 data bytes
64 bytes from aaaa::212:7406:6:606: icmp_seq=1 ttl=63 time=765 ms
64 bytes from aaaa::212:7406:6:606: icmp_seq=2 ttl=63 time=269 ms
64 bytes from aaaa::212:7406:6:606: icmp_seq=3 ttl=63 time=256 ms
64 bytes from aaaa::212:7406:6:606: icmp_seq=4 ttl=63 time=249 ms
64 bytes from aaaa::212:7406:6:606: icmp_seq=5 ttl=63 time=337 ms
^Z
[10]+  Stopped                 ping6 aaaa::212:7406:6:606
user@instant-contiki:~/contiki-2.7/tools$ ping6 aaaa::212:7407:7:707
PING aaaa::212:7407:7:707(aaaa::212:7407:7:707) 56 data bytes
64 bytes from aaaa::212:7407:7:707: icmp_seq=1 ttl=62 time=542 ms
64 bytes from aaaa::212:7407:7:707: icmp_seq=2 ttl=62 time=480 ms
64 bytes from aaaa::212:7407:7:707: icmp_seq=3 ttl=62 time=622 ms
64 bytes from aaaa::212:7407:7:707: icmp_seq=4 ttl=62 time=551 ms
^Z
[11]+  Stopped                 ping6 aaaa::212:7407:7:707
user@instant-contiki:~/contiki-2.7/tools$ ping6 aaaa::212:7408:8:808
PING aaaa::212:7408:8:808(aaaa::212:7408:8:808) 56 data bytes
64 bytes from aaaa::212:7408:8:808: icmp_seq=1 ttl=61 time=763 ms
64 bytes from aaaa::212:7408:8:808: icmp_seq=2 ttl=61 time=904 ms
64 bytes from aaaa::212:7408:8:808: icmp_seq=3 ttl=61 time=831 ms
64 bytes from aaaa::212:7408:8:808: icmp_seq=4 ttl=61 time=858 ms
64 bytes from aaaa::212:7408:8:808: icmp_seq=5 ttl=61 time=748 ms
64 bytes from aaaa::212:7408:8:808: icmp_seq=6 ttl=61 time=863 ms
^Z
[12]+  Stopped                 ping6 aaaa::212:7408:8:808
user@instant-contiki:~/contiki-2.7/tools$ ping6 aaaa::212:7403:3:303
PING aaaa::212:7403:3:303(aaaa::212:7403:3:303) 56 data bytes
64 bytes from aaaa::212:7403:3:303: icmp_seq=1 ttl=62 time=570 ms
64 bytes from aaaa::212:7403:3:303: icmp_seq=2 ttl=62 time=573 ms
64 bytes from aaaa::212:7403:3:303: icmp_seq=3 ttl=62 time=537 ms
^Z
[13]+  Stopped                 ping6 aaaa::212:7403:3:303
user@instant-contiki:~/contiki-2.7/tools$ ping6 aaaa::212:7409:9:909
PING aaaa::212:7409:9:909(aaaa::212:7409:9:909) 56 data bytes
64 bytes from aaaa::212:7409:9:909: icmp_seq=1 ttl=60 time=1049 ms
64 bytes from aaaa::212:7409:9:909: icmp_seq=2 ttl=60 time=1606 ms
64 bytes from aaaa::212:7409:9:909: icmp_seq=3 ttl=60 time=1561 ms
64 bytes from aaaa::212:7409:9:909: icmp_seq=4 ttl=60 time=1668 ms
64 bytes from aaaa::212:7409:9:909: icmp_seq=6 ttl=60 time=1362 ms
```

Figure 3: Pinging results for other nodes

Figure 2 and figure 3 shows the pinging results for the border router and other nodes in the network. TTL value of 60 was observed for the node 9 as shown in figure 3. The datagram sent by the sender is allowed to be in the internet system for certain period of time. This is set by the sender using TTL parameter. If the datagram in the internet system is longer than the TTL value set by the sender, then the datagram must be killed. TTL must be decreased by at least 1 after each hop. If time through the router is less than one second, TTL is decremented by one else it is decremented by two. The time is measured in seconds. The TTL value of 60 thus corresponds to 1 minute. TTL is the upper bound of the time for which the datagram could live in the

network. The idea behind the time bound for the datagram is to cause the undeliverable datagrams to be discarded. This is why the TTL is lower for the nodes than for the border router. [1]

Task 2: Light and Temperature values observation

The second task of this lab was just to observe the temperature and light values simply logging in to the web-server that is running on each nodes. The screenshots from the results observed is shown in the following figures.

Figure 4: Light and temperature reading

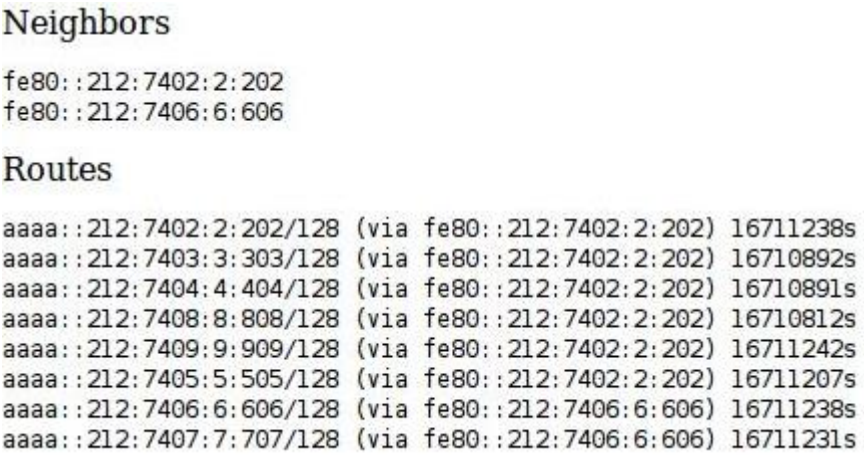

Figure 5: Neighbors and Routes for the border router

Light

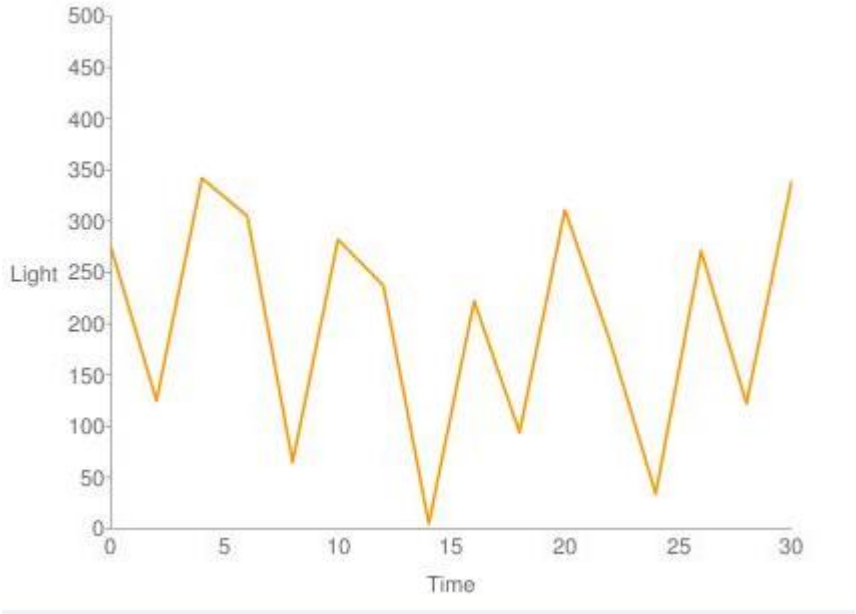

Figure 6: Light readings plotted against time (http://[aaaa::212:7403:3:303]/l)

Temperature

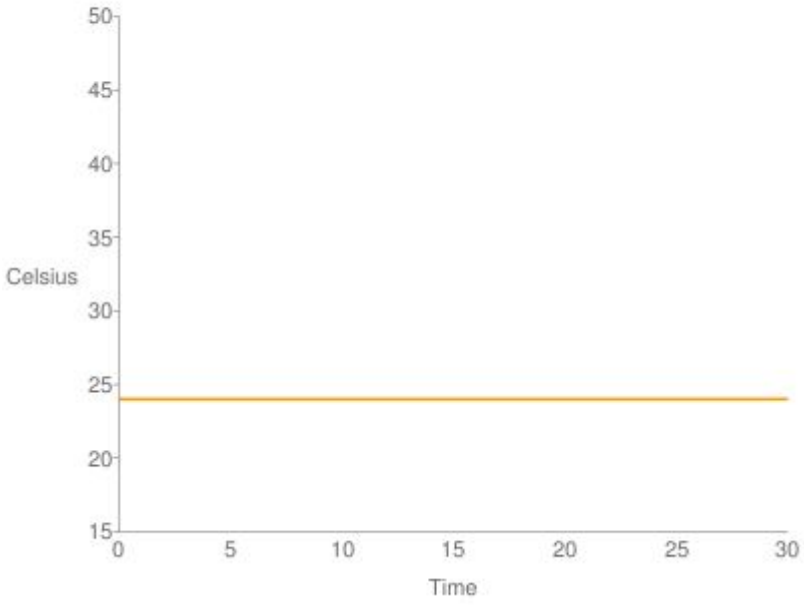

Figure 6: Temperature readings for http://[aaaa::212:7403:3:303]/t

Task 3: Available options for tunslip6

The third task of this lab was to find the available options for tunslip6. Tunslip6 options could also be –B, -S, -T, -V, -d, -a and –p.

-B option is to set up the baud rate for the serial communication, -S is to interface with the serial port, -a for the server address and –p for the serverport.

Task 4: UDP IMPLEMENTATION

The guidelines provided in the lab sheet was followed. At the very first, an RPL border router was created and the port was opened. Tunneling was done using the commands provided. The server address was set to be 2001:5c0:1101:5001::1/64. Two nodes were added to the simulation running udp_client. The host PC was given the address of the server using $ ip -6 addr add 2001: 5c0:1101:5001::1/64 dev eth0 . The server was running simple UDP socket server implementation. Code implementation of the socket server is copy pasted below:

```
/* Sample UDP server */

#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

#define portnumber 6012

int main(void)
{
  int sockfd,n;
  //int portnumber;
  struct sockaddr_in6 servaddr,cliaddr;
  socklen_t len;
  char mesg[1024];


 /* if (argc !=2){
                printf("ERROR: No ports provided\n");

  }*/

  sockfd=socket(AF_INET6,SOCK_DGRAM,0);

  if (sockfd < 0){
                printf("ERROR: Socket could not be created\n");

  }
```

```
 // portnumber=atoi(argv[1]);

  bzero(&servaddr,sizeof(servaddr));
  servaddr.sin6_family = AF_INET6;
  servaddr.sin6_addr=in6addr_any;
  servaddr.sin6_port=htons((unsigned short )portnumber);
  if (bind(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr)) < 0){
    printf("ERROR: Binding failed\n");
  }
  else printf ("SERVER READY\n");

  for (;;)
  {
     len = sizeof(cliaddr);
     n = recvfrom(sockfd,mesg,1024,0,(struct sockaddr *)&cliaddr,&len);
     sendto(sockfd,mesg,n,0,(struct sockaddr *)&cliaddr,sizeof(cliaddr));
     printf("-------------------------------------------------------\n");
     mesg[n] = 0;
     printf("Received the following:\n");
     printf("%s\n",mesg);
     printf("-------------------------------------------------------\n");
  }
}
```

N.B. Refer to my_server.c for the socket udp server program.

The screenshot of the terminal output in the server side is copy pasted here.

```
user@instant-contiki:~/contiki-2.7/examples/ipv6/UTU_IPV6/UDP_server$ ./my_server
SERVER READY
-----------------------------------------------------
Received the following:
Hello 3 from the client
-----------------------------------------------------
-----------------------------------------------------
Received the following:
Hello 3 from the client
-----------------------------------------------------
-----------------------------------------------------
Received the following:
Hello 3 from the client
-----------------------------------------------------
-----------------------------------------------------
Received the following:
Hello 4 from the client
-----------------------------------------------------
-----------------------------------------------------
Received the following:
Hello 4 from the client
-----------------------------------------------------
```

Figure 1: UDP server receiving Hello message from the udp client nodes

Steps followed to achieve this is briefly listed.

- RPL border-router was added
- Opened the serial socket port for listening
- Tunneling in order to bridge the 6lowpan network and external IP network
- Hard-coded the server address in the UDP client program
- Assign the same address to the server running on the local linux machine
- Run the UDP server in the local linux machine
- Run the Cooja simulation

Moreover, the UDP client program was also modified such that the client sends the light and temperature data every 10 seconds. There were couple of things done in order to achieve this. The light and temperature sensing was implemented in the UDP client program. The send packet procedure defined in UDP client was modified such that the procedure now reads the light and temperature data and send it to the UDP server. The light and temperature values were however observed to be constant presumably because of the simulation scenario. There was not any real sensor deployment in the simulation and therefore the data values received were constant all the time. The screen capture of the terminal output is copy pasted below.

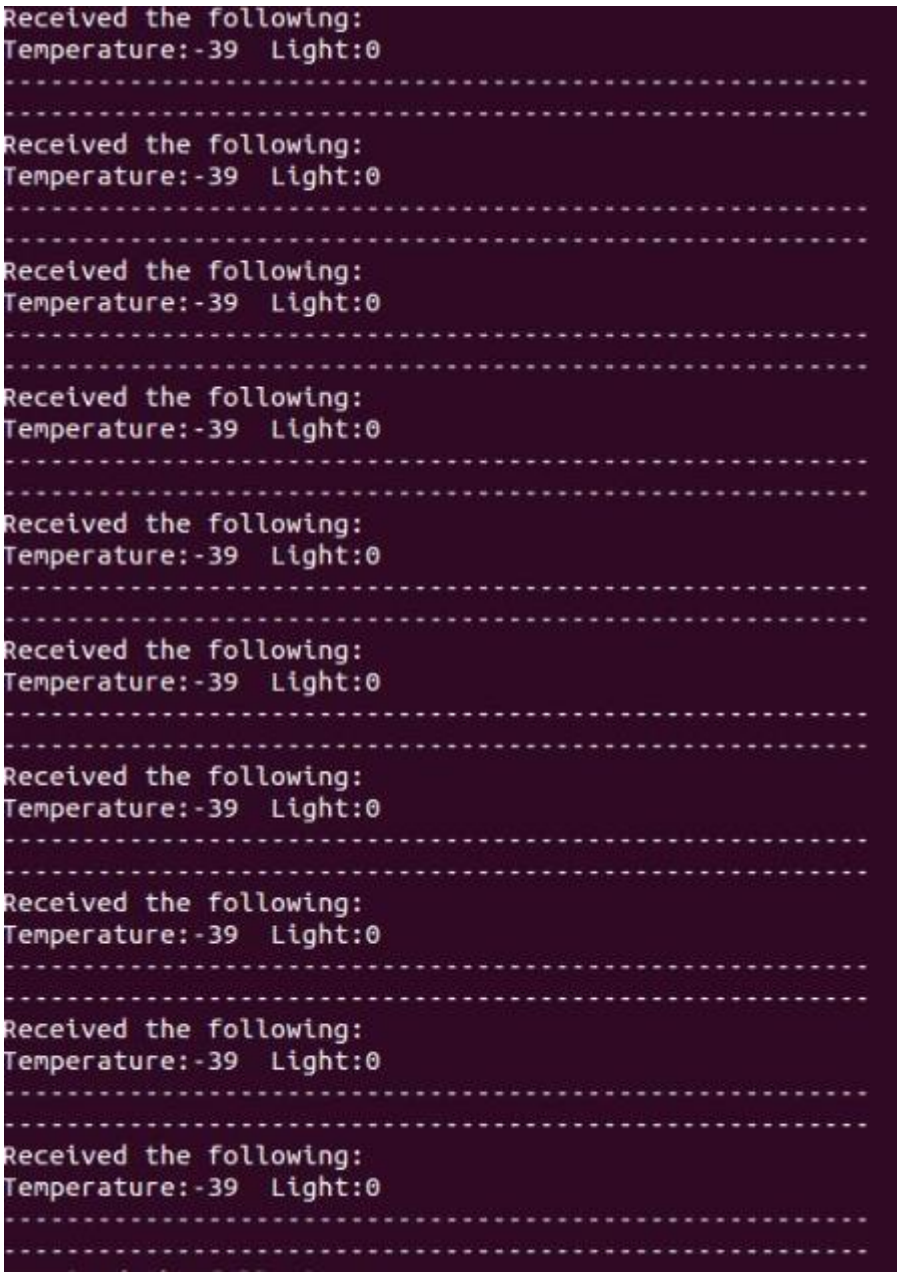

Figure 2: Server terminal output of the light and temperature data

The send_packet procedure was modified such that the procedure first calls the other procedures to read the temperature and light values, copies these read values to a buffer and send the values in the buffer to the server. The modified send_packet procedure is copy pasted here.

```
static void
send_packet(void *ptr)
{
  static int seq_id;
  static int temperature[HISTORY];
  static int light1[HISTORY];
```

```
 static int sensors_pos;
 char buf[MAX_PAYLOAD_LEN];

 light1[sensors_pos] = get_light();
 temperature[sensors_pos] = get_temp();
 //seq_id++;
 PRINTF("DATA sent to %d\n",
      server_ipaddr.u8[sizeof(server_ipaddr.u8) - 1]);
 sprintf(buf, "Temperature:%d  Light:%d", temperature[sensors_pos],light1[sensors_pos]);
 uip_udp_packet_sendto(client_conn, buf, strlen(buf),
                 &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
 sensors_pos+=1;
}
```

The event timer was used for the time synchronization. The client node sends the light and temperature data every 10 seconds. Refer to my-udp-client.c program for the details.

Task 5. Can't we form network without tunnel? Explain the need for tunnel in network?

Tunnel is the mechanism used to transport unsupported protocols over diverse networks. Tunneling is required to bridge the gap between incompatible networks. RPL in WSN creates 6lowpan network. This network is not compatible with IP network (IPv4/IPv6) without adaptation. Tunneling is required such that an adaptation layer is created and hence makes it possible to transport the data packets between the cross networks.

Task 6. Are there any differences between IPv6 and 6lowpan? If yes, explain?

IPv6 is the internet protocol version 6 designed to address the problem of IPv4 protocol i.e. address spaces running short. IPv4 has 32 bits long addresses thereby allowing 2^{32} address spaces. Internet has grown exponentially over the years and the IPv4 address space is saturating. With the introduction of IPv6, the available address spaces is 2^{128} i.e. approximately $3.4 * 10^{38}$ addresses thereby addressing the address space problem. 6lowpan on the other hand is IPv6 protocol analogy designed for low power and lossy networks (LLN). IPv6 and 6lowpan are two different network protocols. IPv6 is designed for regular networks similar to IPv4 while 6lowpan is designed for resource constrained embedded systems.

References

1. http://www.ietf.org/rfc/rfc791.txt