TCP/IP Stack Cheat Sheet

1 Contents

2 INTRODUCTION

This Document describes the APIs and Data structures for tcp/ip stack library built in user-space. This Library simulates the TCP/IP Stack (or alternatively can be called as Network-Emulator)

which has been designed and implemented to practice/demonstrate & implement Networking Algorithms, Networking Protocol Development etc on a Virtual Topology.

Development of this TCP/IP Stack Library is well documented in the form of Complete comprehensive Video Instructor led course here. You must do this Course to implement this TCP/IP Stack Library as a project by yourself in a step by step manner in C on linux platform.

This Document is prepared with the sole objective that would allow one straight away **download and use this tcp/ip stack library without actually understanding the internal implementation of the library**. This document explains the use of tcp/ip stack public APIs which the user of this library can use straightaway to develop networking projects on top of this TCP/IP stack library. This Document serves as the *cheat sheet*.

We shall be going to describe the APIs and relevant Data structures which you would be needing to implement a new Networking assignment (herein referred to as *application*) using this library starting from Physical Layer to Application Layer of the OSI model – in order. Be ready to navigate through the Source code of this library as we navigate through this document.

2.1 About the Project

As we stated, the TCP/IP stack project is a Network-Emulator. Network Emulation is a technique for testing the performance/functionality of real applications over a virtual network. The aim is to assess performance, predict the impact of change, or otherwise optimize technology decision-making.

Network Emulators have been extensively in use for:

1. Proof of concept
2. Performance evaluation

3. Cost Saving
4. Verification and Validation, troubleshooting
5. Fast Development and Deployment
6. Bug reproduction, etc..

This Project is about developing a light weighted network emulator which would support:

1. Quick deployment of Network Custom Topologies
2. Support L2 Routing Protocols (ARP)
3. Support MAC Address Learning and Forwarding (L2 Switching)
4. Support Vlan based Routing
5. Support L3 Routing Over IP
6. Support Auto Learning of Routing Tables using Dijkstra Algorithm
7. Support Statistics collection and extensive logging for Troubleshooting
8. Extensible for Development of Network Applications and Deployment over Existing Infrastructure

The unique-ness of the project is – it provides a clean and fast Network Application Development Environment (NADE) to the Software developer with SDK (Software Development Toolkit) support and testing on virtual topology. The infrastructure provides the virtual environment to the application being developed and testing like on real networking device.

The infrastructure provides L3 IP paths to be used by the application for communication with remote emulated devices over IP protocol. A change in the topology (such as link down/unusable, node failure etc), infrastructure reconverges automatically to compute new L3 paths and continue to provide L3 alternate paths (if available) to the applications.

The project runs as a single multi-threaded process on a single Linux machine. The project is entirely developed in C and provide a command line interface to the user for interactive experience. The project is developed by keeping in mind that it should be open for extension. The developer can develop as many applications as possible using the infrastructure without making any code changes in the infrastructure itself. The Applications are exposed as plugin-in play entities to the infrastructure. The complexity of the internal implementation of the emulator is abstracted away from application developer. Here Infrastructure refers to interconnectivity between nodes, packet exchange simulation and TCP/IP stack mini library implemented.

The Projects do not have GUI and is completely command line based.

Support OS: Linux. Compiler used: gcc, Language C.

Current LOC: 20k approx.

External Libraries used:

- CommandParser for CLI Integration
- LinkedList and Tree Libraries

- Library Wrapper over Linux Timers
- BitOp for bits manipulation

Compiling and Running

2.1.1 Source Code Download

For Non git users: The github link to download src code is here. Download the src code and migrate to your linux machine for building and compiling.

For Git users: Git users can use git clis to download src code using **git clone** cmd, then switch to branch **proto-dev.** I would **strongly recommend** starting using git like version control system if you are not using it already.

```
git clone https://github.com/sachinites/tcpip_stack

cd tcpip_stack

git branch proto-dev
```

Optionally, you should setup some code navigator tool to browse the code. Using Source insight on windows machine is ideal. For that, you would need to host the copy of the code on windows machine as source insight is available only for windows unfortunately. Source Insight is paid software and is expensive. Here is a version 3.5 with key. Download and install from here. Refer to **this Quick Video Tutorial** to setup Source insight to navigate src code of this project. Trust me, it would help you immensely in the long run.

2.1.2 Compiling and building

To build the entire library code, just use **make all** cmd.

```
vm@ubuntu:~/src/tcpip_stack$ make all
make
make[1]: Entering directory '/home/vm/src/tcpip_stack'
gcc -g -c testapp.c -o testapp.o
gcc -g -c -I gluethread gluethread/glthread.c -o gluethread/glthread.o
gcc -g -c -I Tree Tree/avl.c -o Tree/avl.o
gcc -g -c -I . graph.c -o graph.o
gcc -g -c -I . topologies.c -o topologies.o
gcc -g -c -I . net.c -o net.o
gcc -g -c -I . comm.c -o comm.o
gcc -g -c -I . Layer2/layer2.c -o Layer2/layer2.o
gcc -g -c -I . Layer3/layer3.c -o Layer3/layer3.o
gcc -g -c -I . Layer3/netfilter.c -o Layer3/netfilter.o
gcc -g -c -I . Layer4/layer4.c -o Layer4/layer4.o
gcc -g -c -I . Layer5/layer5.c -o Layer5/layer5.o
gcc -g -c -I . nwcli.c  -o nwcli.o
gcc -g -c -I . utils.c -o utils.o
gcc -g -c -I . Layer2/l2switch.c -o Layer2/l2switch.o
gcc -g -c -I gluethread -I libtimer libtimer/WheelTimer.c -o libtimer/Wh
gcc -g   -c -o libtimer/timerlib.o libtimer/timerlib.c
gcc -g -c -I . Layer5/nbrship_mgmt/nbrship_mgmt.c -o Layer5/nbrship_mgmt
gcc -g -c -I . -I Layer5/ddcp/ Layer5/ddcp/ddcp.c -o Layer5/ddcp/ddcp.o
gcc -g -c -I . Layer5/spf_algo/spf.c -o Layer5/spf_algo/spf.o
gcc -g -c tcp_stack_init.c -o tcp_stack_init.o
gcc -g -c -I . tcp_ip_trace.c -o tcp_ip_trace.o
gcc -g -c -I gluethread -I . tcpip_notif.c -o tcpip_notif.o
gcc -g -c -I gluethread -I . notif.c -o notif.o
gcc -g -c -I EventDispatcher -I gluethread EventDispatcher/event_dispatc
gcc -g -c -I . tcp_ip_default_traps.c -o tcp_ip_default_traps.o
gcc -g -c -I . Layer5/isis/isis_adjacency.c -o Layer5/isis/isis_adjacenc
gcc -g -c -I . Layer5/isis/isis_cli.c -o Layer5/isis/isis_cli.o
gcc -g -c -I . Layer5/isis/isis_rtr.c -o Layer5/isis/isis_rtr.o
gcc -g -c -I . Layer5/isis/isis_intf.c -o Layer5/isis/isis_intf.o
gcc -g -c -I . Layer5/isis/isis_pkt.c -o Layer5/isis/isis_pkt.o
gcc -g -c -I . Layer5/isis/isis_events.c -o Layer5/isis/isis_events.o
make[1]: Circular Layer5/isis/isis_flood.o <- Layer5/isis/isis_flood.o d
gcc -g -c -I . Layer5/isis/isis_flood.c -o Layer5/isis/isis_flood.o
gcc -g -c -I . Layer5/isis/isis_lspdb.c -o Layer5/isis/isis_lspdb.o
gcc -g -c -I . Layer5/isis/isis_spf.c -o Layer5/isis/isis_spf.o
(cd CommandParser; make)
make[2]: Entering directory '/home/vm/src/tcpip_stack/CommandParser'
Building testapp.o
Building cmd hier.o
```

After Building, it will produce two executables: **tcpstack.exe** and **pkt_gen.exe**

You need to run **tcpstack.exe** to launch a project. We will talk later about second executable. Second executable is used to generate an IP traffic and feed the traffic into the topology for testing.

Simply run the executable : **./tcpstack.exe**

It will give you **tcp-ip-stack> $** prompt , waiting for user input. TCP/IP stack library provides linux like CLI interface to user for interaction.

Use **show help** to see what default options CLI interface provides to end user. Follow **Help Wizard** to play a bit with CLI interface and understand using it.

Ctrl + C won't help to terminate the running project. Use **run term** command to terminate the project.

```
vm@ubuntu:~/src/tcpip_stack$ ./tcpstack.exe
run - 'show help' cmd to learn more
tcp-ip-stack> $ show help
Parse Success.
Welcome to Help Wizard
======================
1. Use '/' Character after the command to enter
2. Use '?' Character after the command to see po
3. Use 'do' from within the config branch to dir
4. Use '.' Character after the command to see po
5. Built-in commands:
    a. cls - clear screen
    b. cd - jump to top of cmd tree
    c. cd.. - jump one level up of command tree
    d. config [no] console name <console name> -
    e. config [no] supportsave enable - enable/d
    f. debug show cmdtree - Show entire command
    g. show history - show history of commands t
    h. repeat - repeat the last command
                        Author : Abhishek Sagar, J
                        Visit : www.csepracticals.
CLI returned

tcp-ip-stack> $ run term
Parse Success.
Bye Bye
vm@ubuntu:~/src/tcpip_stack$
```

2.1.3 Topology Running

What Topology is the project running when we launch the executable in previous section? All topologies are programmed in file topologies.c . A topologi is built by a topology buid function. To name a few , we have following topologies read made and be launched at will.

Pre-Built topologies:

build_first_topo()

build_simple_l2_switch_topo()

build_square_topo()

build_linear_topo() and few more.

In file **testapp.c** , we have main() fn. Project execution starts from here. Whatever topology is specified in main function **(topo = cross_link_topology ()**), project runs that topology when launched. You can change this particular line with new topology build function, rebuild the project and launch it to load with new topology. Use command: show topology to print the topology.

2.2 Code Organization

The entire source code is present in **tcpip_stack** dir. You can see bunch of src files within the tcpip_stack dir itself (top most level). These files implements common functionality of tcpip_stack which is not bound to a particular layer in OSI Model. There you can see certain directories as well. Those directories contain the source code related to particular functionality. Listing out all top level directories here :

vm@ubuntu:~/src/tcpip_stack$ ls -l | grep ^d

drwxrwxr-x 2 vm vm 4096 Aug 5 04:40 BitOp <<< contain hdr file which provides the macros to work with BIT manipulations.

drwxrwxr-x 2 vm vm 4096 Aug 6 03:34 CommandParser <<< Implement CLI interface

drwxrwxr-x 2 vm vm 4096 Aug 6 03:34 EventDispatcher <<< Implement Scheduler (Event Loop). Used by tcpip_stack core implementation.

drwxrwxr-x 2 vm vm 4096 Aug 6 03:34 gluethread <<< provides doubly linked list implantation called glthreads

drwxrwxr-x 3 vm vm 4096 Aug 6 03:34 Layer2 <<< Implemented Layer 2 specific functionality such as – ARP, MAC learning, VLAN Tagging.

drwxrwxr-x 2 vm vm 4096 Aug 6 03:34 Layer3 <<< Implemented Layer 3 routing

drwxrwxr-x 2 vm vm 4096 Aug 6 03:34 Layer4 <<< Not Supported in our project. Empty directory.

drwxrwxr-x 6 vm vm 4096 Aug 6 03:34 Layer5 <<< Implements Layer 5 common functionality, such as allowing appln to register for packets of interest. Contains code for sample applications.

drwxrwxr-x 2 vm vm 4096 Aug 6 03:34 libtimer <<< A library to Provide an interface to use timer facility

drwxrwxr-x 2 vm vm 4096 Aug 5 22:44 logs <<< Dir where are logging output is written.

drwxrwxr-x 2 vm vm 4096 Aug 5 12:03 ted <<< Traffic engineering Database library. More on this later

drwxrwxr-x 2 vm vm 4096 Aug 6 03:34 Tree <<< Provides AVL tree library interface.

2.3 Base Data Structures

2.4 Base APIs

We need some basic APIs to work with TCP/IP stack library. The APIs which you would be needing to use in this project are discussed below :

2.4.1 Getting a node ptr from Node name

While writing CLIs , we would need to get the pointer to the node which represents a device in the topology. We can obtain node ptr using below API. ‘topo’ is a global variable which represents a network topology instance.

```
node_t *node = get_node_by_node_name(topo, node_name);
```

2.4.2 Getting a interface ptr from interface name

While writing CLIs , we would need to get the pointer to the interface of the device identified by interface name. We can obtain interface_t ptr using below API.

interface_t *intf = node_get_node_by_name(node, if_name);

where *node* is a pointer to the node_t data structure obtained from API discussed in 1.3.1

2.4.3 Looping over all interfaces of a devices

Many times, we would feel the need to loop over all the interfaces of a device. We can use below code to do:

```
int i = 0;
  for(; i < MAX_INTF_PER_NODE; i++){
      intf = node->intf[i];
      if ( !intf ) continue;
    /*Do your processing */
}
```

3 BUILDING UP NEW NETWORK TOPOLOGY

4 PHYSICAL LAYER

Physical Layer APIs would consist of all APIs required to send frames/packets directly on a physical interface of a device by the application.

1. This API is used to send the packet *pkt* of size *pkt_size* in bytes out of the local interface *interface* of a device.

int send_pkt_out (char *pkt, /*ptr to the pkt*/

uint32_t pkt_size, /*pkt size*/

interface_t *interface); /*O I F*/

@return int - No of bytes sent, return -ve value in case of error

2. This API is used to send the packet *pkt* of size *pkt_size* bytes out of all local interfaces of a device except the exempted interface *exempted_intf.*

   int send_pkt_flood (node_t *node,

   interface_t *exempted_intf,

   char *pkt, uint32_t pkt_size);

   @return int - No of bytes sent, return -ve value in case of error

   We will discuss in Transport Layer Section how an Application Can receive packets.

4.1 Interface Properties

A Device have multiple interfaces as shown below. The device can be Router or a Switch or Both.

Here we discuss some generic properties of an interface of a Network Device and APIs to manipulate or access interface properties. An interface is represented by structure *interface_t* defined in ***graph.h*** and interface generic properties are defined in *intf_nw_props_t* defined in ***net.h***.

4.1.1 L3 Interface Mode

An interface of a Device can operate either in L2 mode or in L3 mode. As an application developer, we need to bother only about interfaces operating in L3 mode. An Interface is said to be operating in L3 mode if it is configured with IP address.

To check if a Given interface *interface_t* is operating in a L3 mode :

IS_INTF_L3_MODE(interface_ptr)

@returns boolean true if interface is configured with IP address, otherwise false.

4.1.2 L2 Interface Mode

An Interface is said to be operating in L2 mode if it is either configured to operate in **ACCESS** mode or in **TRUNK** mode. To get the interface L2 mode value, use

IF_L2_MODE(interface_ptr)

@returns L2 mode value which is enum defined in net.h as ACCESS , TRUNK or L2_MODE_UNKNOWN.

Note: An interface operate either in L2 mode or L3 mode but not both at a time.

4.1.3 Get Interface IP Address, Mask and Mac Addr

If interface is Operating in L3 mode, then you can get IP address and Mask configured on an interface.

char *ip = **IF_IP(interface_ptr)**

char mask = **IF_MASK(interface_ptr)**

Interface always have MAC address . no matter if interface is in L3 mode or L2 mode. Mac can be obtained as

char *mac = **IF_MAC(interface_ptr)**

4.2 Device Properties

5 DATA LINK LAYER

Data Link Layer Involves the Layer 2 Protocol Handling. TCP/IP Stack Library implements the simple mini ARP (Address Resolution Protocol). At Data link Layer, You would mainly deal with Ethernet Hdr and ARP hdr.

5.1 Ethernet Header

The Ethernet Hdr structure is described in **Layer2/layer2.h**

```
typedef struct ethernet_hdr_{
    mac_add_t dst_mac;
    mac_add_t src_mac;
    unsigned short type;
    char payload[248];  /*Max allowed 1500*/
    uint32_t FCS;
} ethernet_hdr_t;
```

ethernet_hdr_t

Max payload allowed is 248B. You need to increase this limit in case your requirement is to prepare bigger ether packets.

5.1.1 Mac Address

The Mac Address is represented by:

```
typedef struct mac_add_ {
    unsigned char mac[6];
} mac_add_t;
```

5.1.2 Getting the Size of Eth hdr Excluding Payload

GET_ETH_HDR_SIZE_EXCL_PAYLOAD This Macro represents the Size of **Dst Mac + Src Mac + Type + FCS** fields in Ethernet hdr which is 18B (Yellow Region) and is Constant.

5.1.3 Getting and Setting the Value of Ethernet hdr FCS

GET_COMMON_ETH_FCS – This macro returns the value of FCS field in the ethernet hdr which is 4B

SET_COMMON_ETH_FCS – This macro sets of value of FCS field in the ethernet hdr which is 4B

5.1.4

5.1.5 Malloc-ing a new Ethernet Packet

Once you know the size of Ethernet Payload *payload_size* to be carried by Ethernet Hdr, you can malloc the ethernet pkt in your application as below:

```
ethernet_hdr_t *ethernet_hdr = (ethernet_hdr_t *)
tcp_ip_get_new_pkt_buffer (ETH_HDR_SIZE_EXCL_PAYLOAD
+ payload_size);
```

5.1.6 Free a Packet Memory

```
static inline void tcp_ip_free_pkt_buffer(char *pkt, uint32_t
pkt_size);
```

5.1.7 Get the Pointer to the Eth-Payload from Ethernet Hdr

If you are given pointer to the ethernet hdr, then pointer to the encapsulated payload can be obtained as

```
unsigned char *payload =
GET_ETHERNET_HDR_PAYLOAD(ethernet_hdr_ptr)
```

Dst_mac (6B)	Src_mac (6B)	Type (2B)	Payload (.. 248)	FCS (4B)

^ << Returns the start of the payload

5.1.8 Updating the FCS of Ethernet Hdr

The below Macro can be used to update the FCS field of the given ethernet hdr. Our TCP/IP stack do not manipulate this value, so you can always update it with zero.

```
SET_COMMON_ETH_FCS(eth_hdr_ptr,     /* Pointer to the Ethernet
ethernet_hdr_t */
                   payload_size,    /*Size of Payload container in
ethernet hdr*/
                   value)           /*FCS new Value to set*/
```

Use:

```
SET_COMMON_ETH_FCS(ethernet_hdr_ptr, payload_size, 0);
```

Don’t do: Do not update the FCS of ethernet hdr directly as below, it will lead to pkt Corruption.

```
ethernet_hdr_ptr->FCS = <value> ✘
```

5.1.9 Getting the size of Ethernet Payload

To get the size of payload contained within the ethernet hdr and given that you know the total size of the packet which is *total_pkt_size*:

```
uint32_t payload_size = total_pkt_size -
                        GET_ETH_HDR_SIZE_EXCL_PAYLOAD(eth_hdr_ptr);
```

5.1.10 Filling the Dst Mac with Broadcast Mac

In case your Application need to prepare the ethernet hdr and want to fill the dst mac with broadcast mac address FF:FF:FF:FF:FF:FF , then you can use below API –

void layer2_fill_with_broadcast_mac(char *dst_mac);

For any other value other than Broadcast MAC, you need to fill the dst mac (or src mac) programmatically.

Use :

```
layer2_fill_with_broadcast_mac(ethernet_hdr->dst_mac.mac);
```

5.1.11 Checking if Mac Address is Broadcast MAC

Library provides a macro which evaluates whether the given mac address is Broadcast mac or not.

```
IS_MAC_BROADCAST_ADDR(mac)
```

This macro is implemented in **tcpip_stack/utils.h**

Use :

```
IS_MAC_BROADCAST_ADDR(ethernet_hdr->dst_mac.mac)
```

5.1.12 Miscellaneous

- Rest of the Fields of the Ethernet Hdr you can manipulate directly if required in your application.
- If you need to write a new API manipulating Ethernet hdr, write code in Layer2/layer2.c|h

○ ARP Table and ARP Resolution

ARP table of the device is populated automatically via ARP resolution. To check ARP table of the device use cmd:

show node <node-name> arp

Example:

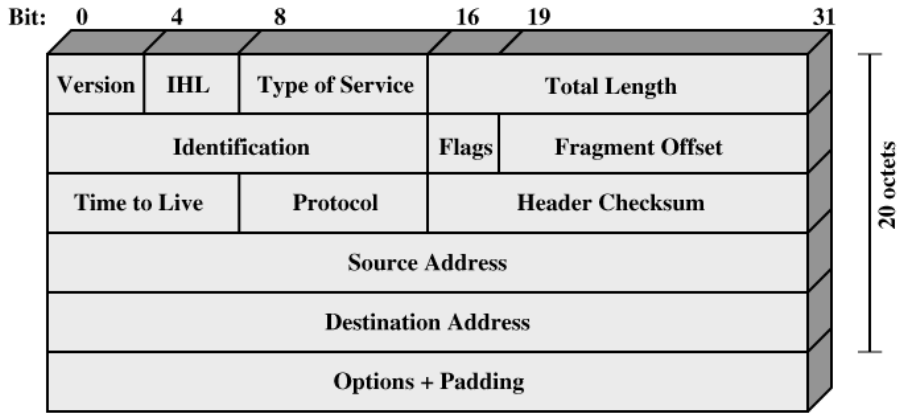
```
tcp-ip-project> $ show node R2 arp
Parse Success.
        |========IP==========|========MAC========|=====OIF======|===Resolved=====|
        | 10.1.1.1           | f3:19:cf:88:00:00 |  eth1        |   TRUE         |
        |====================|===================|==============|================|
```

6 NETWORK LAYER

The Network Layer of TCP/IP stack primarily deals with Routing. Our TCP/IP Stack implements IP routing as a Network Layer Routing Protocol.

6.1 IP Header

At Network Layer, We would need to deal with IP Phdr (without options).



TCP/IP stack library defines IP hdr as **ip_hdr_t** in **Layer3/layer3.h** file as described below. Only main fields which are essential to perform IP routing are supported (highlighted in green below) by tcp/ip stack library. Fields which are in red color can

always be left out with default or zero values. The size of IP Hdr without option field is 20B.

```
/*The Ip hdr format as per the standard specification*/
typedef struct ip_hdr_{
    uint32_t version : 4 ;  /*version number, always 4 for IPv4 protocol*/
    uint32_t ihl : 4 ;      /*length of IP hdr, in 32-bit words unit. for Ex, if this value is 5, it means length of this ip hdr is 20Bytes*/
    char tos;
    short total_length;         /*length of hdr + ip_hdr payload*/
    /* Below are Fragmentation Related members, we shall not be using below members
     * as we will not be writing fragmentation code. if you wish, take it as a extension of the project*/
    short identification;
    uint32_t unused_flag : 1 ;
    uint32_t DF_flag : 1;
    uint32_t MORE_flag : 1;
    uint32_t frag_offset : 13;
    char ttl;
    char protocol;
    short checksum;
    uint32_t src_ip;
    uint32_t dst_ip;
} ip_hdr_t;
```

Now Let us discuss APIs to work with Ip Hdr.

6.1.1 Initializing IP Hdr

IP Hdr can be initialized with all fields set to defaults using below API defined in Layer3/layer3.h. Application must malloc the memory for IP hdr and call this API to initialize it. After that you must over-write appropriate fields of IP hdr.

```
static inline void initialize_ip_hdr ( ip_hdr_t *ip_hdr );
```

Use :

```
ip_hdr_t iphdr;

initialize_ip_hdr(&iphdr);
```

6.1.2 Calculating total_length field of IP Hdr

IP Hdr contains the field total_length which is the length of IP hdr + IP payload expressed in a 32bit units. For examples, if the size of IP hdr + payload is 62B, then total_length should be

Size of IP Hdr in 4B unit + Size of Payload in 4B unit + 1 extra 4B unit for round off

(short)(20)/4 + (short)(62 – 20)/4 + (shirt)(62 % 4) ? 1 : 0 = 16

You should initialize the *total_length* field of IP hdr as below. We have provided a below Macro to do above math for you. You need to pass the size of IP hdr Payload in bytes.

```
iphdr.total_length = IP_HDR_COMPUTE_DEFAULT_TOTAL_LEN(ip_payload_size);
```

6.1.3 Get Len of IP Hdr

Given that you have a pointer to IP Hdr, Ip Hdr len in Bytes can be obtained as :

```
uint32_t len = IP_HDR_LEN_IN_BYTES(ip_hdr_ptr)
```

This macro returns 20B assuming IP hdr to not have ‘options’ field.

6.1.4 Get Len of IP Hdr + IP Payload

Given that you have a pointer to IP Hdr, len of IP Hdr and IP payload can be obtained as :

```
uint32_t len = IP_HDR_TOTAL_LEN_IN_BYTES(ip_hdr_ptr)
```

6.1.5 Get IP Hdr Payload

Given that you have a pointer to IP Hdr, Pointer to IP Payload can be obtained as :

```
char *payload = INCREMENT_IPHDR(ip_hdr_ptr)
```

This payload data could be transport header or application data.

6.1.6 Get IP Payload Size

Given that you have a pointer to IP Hdr, Size of IP Payload in Bytes can be obtained as :

```
uint32_t size = IP_HDR_PAYLOAD_SIZE(ip_hdr_ptr)
```

6.2 Sending the Application Data to Remote Device

Once the Application has its data ready, it can send the data to remote host using destination IP address. API to send data is:

```
void
tcp_ip_send_ip_data (node_t *node,                  /*Sending Node*/
                     char *app_data,                /*Ip Payload*/
                     uint32_t data_size,            /*Size of IP payload in Bytes*/
                       int L5_protocol_id,          /*Appln Protocol No*/
                     uint32_t dest_ip_address)    /*Destination IP address */
```

6.3 L3 Routes and L3 Routing Table

The TCP/IP stack library computes L3 Routes for all routers in the topology automatically and install in L3 routing table of L3 enabled devices as soon as topology is launched. These routes are computed as per the shortest path first algorithm. Via these auto computed L3 routes, each device in the topology is reachable from every other device via shortest path. There shall be no loops.

To check routing table on a device use cmd :

show node <node-name> rt

Example:

```
tcp-ip-project> $ show node R1 rt
Parse Success.
L3 Routing Table:
        |======= IP ========|== M ==|======== Gw ========|===== Oif ====|
        |122.1.1.2          |  32    | 10.1.1.2           | eth0         |
        |===================|=======|====================|==============|
        |122.1.1.4          |  32    | 40.1.1.1           | eth7         |
        |===================|=======|====================|==============|
        |122.1.1.3          |  32    | 10.1.1.2           | eth0         |
        |                   |        | 40.1.1.1           | eth7         |
        |===================|=======|====================|==============|
        |40.1.1.0           |  24    | NA                 | NA           |
        |===================|=======|====================|==============|
        |10.1.1.0           |  24    | NA                 | NA           |
        |===================|=======|====================|==============|
        |122.1.1.1          |  32    | NA                 | NA           |
        |===================|=======|====================|==============|
```

For local routes, **Gw** and **Oif** are **NA** (Not applicable)

As a user, if you want to install an additional route manually in Routing table of a device, then you can use cmd :

config node <node-name> route <ip-address> <mask> <gw-ip> <oif>

Eg : config node R1 route 122.1.1.1 32 10.1.1.1 eth1

Note : L3 Routes will not be computed along the path of L2 switches if present in the topology.

7 APPLICATION LAYER

Wondering Where transport Layer gone? Our TCP/IP Stack library do not support Transport Layer. It would mean – Data transmission from one device to other in the network happens directly via Layer 3 IP protocol. There is no UDP/TCP traffic. Not having a Transport Layer doesn't restrict us from implementing Networking Algorithms Or Networking problem statement – unless the problem statement demands the transport layer support explicitly.

All Application Code would go in **Layer5/<appln>** dir. For example, we have implemented the example application **ddcp – Distributed Data Collection Protocol** and its code is present in **Layer5/ddcp** dir. You must create as many folder as many applications you want to implement in Layer5 dir only.

We shall going to discuss the ddcp protocol implementation in detail as an example to show how to use TCP/IP Stack library to solve Networking Problems.

7.1 APIs to send Data by the Application

When application wishes to send data to the remote device, it has to perform basic two functions:

1. Prepare a data in the data buffer
2. Send the data

Here, depending on problem statement, Appn may wish to send data :

a. Out of the specific interface to the direct nbr

Use : send_pkt_out(. . .) / send_pkt_flood(…) APIs

b. Outbound to specific Destination in the network

Use : tcp_ip_send_ip_data(. . .)

7.2 STEPS TO WRITE A NEW APPLICATION

You may want to write a new application or Network Protocol which will work on top of our TCP/IP stack library.

All Application shall be written in **tcpip_stack/Layer5/<app>** folder where <app> is the application folder where all src files implementing the application shall be placed. For demonstration purpose, we have implemented the application DDCP (Distributed Data Collection Protocol). It is not a standard application, do not try to google it. All src codes of the application are present in **tcpip_stack/Layer5/ddcp** directory. Let me discuss the steps taking DDCP as a case study regarding how you should write a new application on top of our TCP/IP stack. Implementation of DDCP is discussed in detail through a full-fledge online course.

7.2.1 Step 1 : Defining new Protocol Numbers

DDCP application defines two new protocols:

DDCP_MSG_TYPE_FLOOD_QUERY 1 /*Randomly chosen, should not exceed 2^16 -1*/

DDCP_MSG_TYPE_UCAST_REPLY 2 /*Randomly chosen, must not exceed 255*/

The new protocol Numbers defined by the application can be #defined in **tcpip_stack/tcpconst.h**. Take a look. Take care that protocol numbers you have chosen must not overlap or already taken. Now, how these protocol numbers are used and what is the functionality of these protocol is out of the scope of this document. Just be reminded, if you need to define new protocol numbers, protocol numbers must be defined in the said header file. Be noted that first protocol is the Data link layer protocol (like ARP), and second protocol is Layer 3 Or network Layer protocol. The first protocol shall go in *type* field of ethernet hdr while second protocol shall go as *protocol* field in IP hdr.

7.2.2 Step 2 : Writing new Src files

If you are writing a new protocol functionality, it is evident that you shall be creating a new src files. The DDCP functionality is implemented in ddcp.c and ddcp.h file.

```
vm@ubuntu:~/Documents/tcpip_stack/Layer5/ddcp$ ls -l
total 36
-rw-rw-r-- 1 vm vm 21487 Jun 29 12:19 ddcp.c
-rw-rw-r-- 1 vm vm  5167 Jun 23 12:09 ddcp.h
-rw-rw-r-- 1 vm vm  3031 Jun 23 12:09 serialize.h
```

In all src files you create, you must include **#include "../../tcp_public.h".** This fill import all TCP/IP stack APIs and data structures that you would be working with in ddcp.c.

Whatever new src files you define, you need to update the project Makefile below to include the new src file(s) for compilation.

tcpip_stack/Makefile

Since we have defined ddcp.c as new src file, we shall add an entry in Makefile at three places as below:

1.

```
OBJS=gluethread/glthread.o \
        . . . . .
        . . . . .
        Layer4/layer4.o  \
        Layer5/ddcp/ddcp.o \
        Layer3/spf.o      \
        . . . . . .
```

tcpip_app_register.c

2.

Layer5/ddcp/ddcp.o:Layer5/ddcp/ddcp.c

${CC} ${CFLAGS} -c -I . -I Layer5/ddcp/ Layer5/ddcp/ddcp.c -o Layer5/ddcp/ddcp.o

3.

clean:

rm -f *.o

. . . . .

rm -f Layer4/*.o

rm -f Layer5/*.o

rm -f Layer5/ddcp/*.o

rm -f WheelTimer/WheelTimer.o

7.2.3 Step 3 : Protocol Interest Registration

As stated previously, ddcp application defines two new protocols. Our TCP/IP stack would not know what to do when it receives a packet with these protocol IDs in type or protocol field of ethernet hdr or IP hdr. Hence, our application needs to tell TCP/IP stack library an action to perform when TCP/IP running on a node running ddcp application receives such packet from other nodes in the network.

Here the ddcp behavior is:

1. Whenever the TCP/IP stack receives the frame with *type* field value as DDCP_MSG_TYPE_FLOOD_QUERY, then Data link layer must remove the ethernet hdr and handover the payload part of ethernet hdr to the ddcp application for processing.
2. Whenever the TCP/IP stack receives the packet with *protocol* field as DDCP_MSG_TYPE_UCAST_REPLY in IP Hdr of the packet, then Layer3 of the TCP IP stack must handover the IP hdr payload to ddcp application for further processing. This handover should be done only at Destination machine/device (i.e. the IP packet has reached its intended destination). This IP packet is forwarded via L3 routing across intermediate L3 routers as usual.

So, here, ddcp application needs to program the underlying TCP/IP stack to handover the packet with these protocol IDs to the ddcp protocol running in application layer. How to achieve this ? Lets discuss the steps.

7.2.3.1 Step3.1 Writing a protocol init function

When topology is loaded and start running, we may need to initialize our user defined ddcp protocol to perform one time initialization. In this case this initialization means – telling underlying TCP/IP stack the protocol interest.

We will write an API :

void init_ddcp(){ . . . } in ddcp.c. This API needs to be invoked from below function which is defined in tcp_stack_init.c.

```
void init_tcp_ip_stack(){
   compute_spf_all_routers(topo);
   init_ddcp();
}
```

7.2.3.2 Step3.2 Application Protocol Registration

Now let us discuss how ddcp protocol can express its protocol interest with TCP/IP stack library. This Step is equivalent to opening a sockets on actual TCP/IP socket. In real code base, you wont open a socket directly to allow your application to receive pkt from network. We often have high level APIs using which application protocol program the underlying networking infratrusture/TCP-stack to express interest that in what type of packets the application is interested in.

You need to write the init API as below to begin application protocol registration with underlying TCP/IP stack library. See below code.

```
void init_ddcp(){
   tcp_app_register_l2_protocol_interest(DDCP_MSG_TYPE_FLOOD_QUERY,
      ddcp_process_ddcp_query_msg);
   tcp_app_register_l3_protocol_interest(DDCP_MSG_TYPE_UCAST_REPLY,
      ddcp_process_ddcp_reply_msg);
}
```

You can see above, ddcp protocol is telling underlying TCP/IP stack to invoke the ddcp specific function *ddcp_process_ddcp_query_msg* whenever TCP/IP stack received ethernet pkt with protocol id specified as DDCP_MSG_TYPE_FLOOD_QUERY. TCP/IP stack will invoke the ddcp specific function ddcp_process_ddcp_query_msg and pass the frame's payload as argument to this function. The ddcp function prototype has to be :

void <fn_name> (node_t *, interface_t *, char *, uint32_t, uint32_t);

For example:

```
void
ddcp_process_ddcp_query_msg ( node_t *node,    /*Current node in the Network*/
                interface_t *iif,                    /*The Receiving interface*/
                    char *pkt,                       /*ptr to the start of the packet*/
                uint32_t pkt_size,           /*pkt size*/
                    uint32_t flags);                 /*flags – described  shortly*/
```

Similarly, via Second API tcp_app_register_l3_protocol_interest( ) , application ddcp is conveying underlying TCP/IP stack to invoke the ddcp function ddcp_process_ddcp_reply_msg( ) in case the IP packet has protocol field value as DDCP_MSG_TYPE_UCAST_REPLY and IP pkt has reached its intended destination. The prototype of this fn is also same as previous.

Example:

```
void ddcp_process_ddcp_reply_msg (node_t *node,  /*Current node*/
                                  interface_t *recv_intf, /*intf on which pkt is
receeieved*/
                                    char *pkt,            /*ptr to the IP payload*/
                                  uint32_t pkt_size,      /*IP payload size*/
                                   uint32_t flags);       /*Flags – described next*/
```

The implementation of both the ddcp functions shall go in one of ddcp src files – in this case ddcp.c. Depending on the application functionality and problem statement, the application can process the packet. For example, the ddcp application when receives ethernet payload of type DDCP_MSG_TYPE_FLOOD_QUERY, ddcp application builds up some local database based on packet contents and decides to further flood the packet on other interfaces of the router.

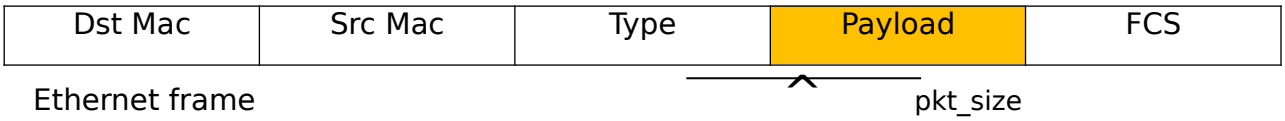
7.2.3.3 Step3.3 Flags

In Step3.2 , you noticed that last argument of ddcp APIs for packet reception from underlying Networking Infrastructure had flags as an additional argument. This flag tells the application that actually what packet it has received from underlying TCP/IP stack layers.

For example, when DDCP protocol receives the ethernet payload for ethernet frames of type DDCP_MSG_TYPE_FLOOD_QUERY, then by default TCP/IP stack remove the ethernet hdr from the frame and convey only the payload portion of the frame to ddcp protocol.

```
ddcp_process_ddcp_query_msg ( node_t *node,    /*Current node in the Network*/
                interface_t *iif,                    /*The Receiving interface*/
                    char *pkt,                       /*ptr to the start of the packet*/
                uint32_t pkt_size,                   /*pkt size*/
                    uint32_t flags);                 /*flags – described  shortly*/
```

It would simply means – the *pkt* pointer (3rd arg)would point to the start of the payload of ethernet frame, and pkt_size would be the size of payload.



The value of flag in this case shall be default which is 0.

However, there could be scenarios where application would be interested in receiving the complete ethernet frame – that is ethernet payload along with ethernet hdr. In this case, application has to modify the default behavior of TCP/IP stack (to remove the ethernet hdr). Application can do so using below API :

void tcp_ip_stack_register_l2_proto_for_l2_hdr_inclusion (uint32_t L2_protocol_no);

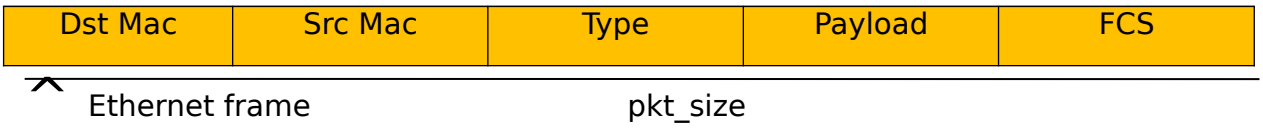
Example : Copy-pasting the init_ddcp(..) again below.

```
void init_ddcp(){
   tcp_app_register_l2_protocol_interest(DDCP_MSG_TYPE_FLOOD_QUERY,
      ddcp_process_ddcp_query_msg);
```

```
tcp_ip_stack_register_l2_proto_for_l2_hdr_inclusion(DDCP_MSG_TYPE_FLOOD_QUERY
);

    tcp_app_register_l3_protocol_interest (DDCP_MSG_TYPE_UCAST_REPLY,

        ddcp_process_ddcp_reply_msg);

}
```

You can see above, ddcp is conveing the TCP/IP stack to handover the frame along with ethernet hdr as it is to the ddcp application whenever the type field in the frame is DDCP_MSG_TYPE_FLOOD_QUERY.

In this case the *flags* value shall be *DATA_LINK_HDR_INCLUDED* which is 2 as defined in tcpconst.h. The *pkt* pointer shall be set at the start of dst mac and *pkt_size* shall be set to the size of entire frame received as show in below diagram.



It is evident, if next hdr in the frame is IP hdr (or whatever) , then IP hdr shall also be shipped as part of ethernet payload.

Similarly, Application may be interested not in ethernet hdr, but interested in IP hdr only succeeding ethernet hdr in the frame. Application can do so using :

```
tcp_ip_stack_register_l3_proto_for_l3_hdr_inclusion(DDCP_MSG_TYPE_UCAST_REPLY);
```

Using above call, ddcp is telling TCP/IP stack to not to remove IP hdr from the those IP packets in which protocol field value in IP hdr is DDCP_MSG_TYPE_UCAST_REPLY, and handover the packet along with IP hdr and IP hdr payload to the application. In this case, the value of *flags* shall be IP_HDR_INCLUDED which is defined as 1 in tcpconst.h.

Application must check these flags as to know what it has actually received from TCP/IP stack and accordingly typecast the *pkt* pointer and process the packet.

7.2.3.4 Step3.4 Defining Application Data Structures

Almost all Networking Application needs to define its own Data structures to implement its business logic. These Data structures are defined as two Levels – The Device Level and Device's interface Level. The Application Protocol Config as device level impact or change the protocol behavior at the device level. For example, disabling the ddcp protocol at the Router R0. Other set of protocol configuration is maintained per interface level. Config at interface level impact the protocol

behavior on that particular interface of the device. For example, disabling sending out ddcp queries out of interface eth0 of the device.

So, whenever we write or implement a new Networking protocol, we need to define the Application/protocol specific data structures and hook them up at node and interface level. Let us continue with ddcp example.

In ddcp.h we define two ddcp related data structures:

ddcp_db_t >> This is device level data ddcp structure.

and

ddcp_interface_prop_t >> this is per interface level ddcp data structure.

We will add device level ddcp data structure member pointer in *node_nw_prop_t* data structure.

```
typedef struct node_nw_prop_{
. . .
. . .
ddcp_db_t *ddcp_db;
. . .
} node_nw_prop_t;
```

The above ddcp_db structure either must be initialized in fn init_node_nw_prop(. . .). The init_node_nw_prop() fn is called when topology is loaded.

```
init_node_nw_prop(node_nw_prop_t *node_nw_prop) {
    . . .
    . . .
    init_ddcp_query_db(&(node_nw_prop->ddcp_db));
    . . .
}
```

This approach assumed that ddcp protocol state always needs to be maintained on all devices by default. If you chose to not to initialize the ddcp_db structure by default (i.e. on every device when topology is loaded), then user must control the enable or disable of ddcp protocol at device level through CLI something link:

config node R0 ddcp enable

which will init/deinit the structure at device level.

We will add per interface level ddcp data structure member pointer in *intf_nw_props_t* data structure.

```
typedef struct intf_nw_props_ {
. . .
. . .
ddcp_interface_prop_t *ddcp_interface_prop;
. . .
} intf_nw_props_t;
```

The above ddcp_interface_prop structure either must be initialized in fn init_intf_nw_prop (. . .). The init_intf_nw_prop () fn is called for every interface of a device when topology is loaded.

```
init_intf_nw_prop(intf_nw_props_t *intf_nw_props){
. . .
. . .
init_ddcp_interface_props(&intf_nw_props->ddcp_interface_prop);
. . .
}
```

Again, this approach assumed that ddcp protocol state always needs to be maintained on all interfaces of the device by default. If you chose to not to initialize the ddcp interface level state by default on all interfaces of the device (i.e. on every device's interface when topology is loaded), then user must control enable or disable of ddcp protocol at interface level through CLI something link:

config node R0 interface eth0 ddcp enable

which will init/deinit the ddcp state per interface level.

Needless to say:

1. If ddcp protocol is not enabled at device level, user must not be able to enable ddcp or configure ddcp at interface level.
2. If ddcp protocol is disabled at device level, then ddcp interface level state must be cleaned up on all interfaces of the device.

This is common practice that all Networking protocols abide by.

Note that the init fn for ddcp protocol are defined in ddcp.c:

```
void init_ddcp_query_db(ddcp_db_t **ddcp_db);
```

```
void init_ddcp_interface_props(ddcp_interface_prop_t **ddcp_interface_prop);
```

To ensure Header file independency, provide forward declaration of Application structure in net.h file before using the member pointer of application structure. Do not #include any application specific header file (= ddcp.h) in net.h file.

Example :

In net.h, below forward declarations are added before using ddcp structures.

```
typedef struct ddcp_db_ ddcp_db_t;

typedef struct ddcp_interface_prop_ ddcp_interface_prop_t;
```

and below functions are imported in ddcp.h before invoking them. This is a common pattern to be followed for all applications.

```
extern void init_ddcp_query_db(ddcp_db_t **ddcp_db);

extern void init_ddcp_interface_props(ddcp_interface_prop_t **ddcp_interface_prop);
```

8 DEBUGGING

Without having a proper Debugging Infrastructure, it would be nightmare to debug what went wrong while testing the application protocol. In Networking, one of the most common way of debugging the networking application is to inspect the pkt contents being flowing across the network topology. Tcpdump, Wireshark, pcap are some common tools used in Networking for analyzing packet content. For our TCP/IP stack library, we have provided a bunch of Commands using which you can inspect the packet content moving across devices in the topology. Since our TCP/IP and topology is a simulation software, standard tools as listed above shall not work with ease with our TCP/IP stack library.

The Three debugging CLIs are provided using which you can inspect the packet content.

1. config global stdout (alternate : ctrl + C)
2. config node <node-name> traceoptions flag <flag-val>
3. config node <node-name> interface <if-name> traceoptions flag <flag-val>

possible values for *flag-val* for CLI 2 and 3 are:

: all | no-all

: recv | no-recv

: send | no-send

: stdout | no-stdout

8.1 Packet Capture

Whenever logging is enabled, packet content is always written to **tcpip_stack/logs** directory. Log files are created which have names like <device_name>.txt and <device_name-intf_name>.txt.

For example, H1.txt file will show all packets being received or being sent on any interface of host/device H1.

Whereas, H1-eth0.txt will show all packets being received or being sent on a specific local interface eth0 of device H1.

Note that , in separate terminal window, you can always do “tail -f <filename.txt>” to see logs as soon as they are written to a file.

The CLI # 2 is used to enable logging at device level, whereas CLI# 3 is used to enable logging for a particular interface of a device.

Examples:

config node H1 traceoptions flag all

Enable send and recv logging on all interfaces of device H1. All Logs will go in H1.txt

config node H1 traceoptions flag no-all

Disable all logging at device level as well as for all interfaces on device H1.

config node H1 traceoptions flag recv

Enable only recv logging on all interfaces of device H1. All Logs will go in H1.txt

config node H1 traceoptions flag no-recv

Disable only recv logging on all interfaces of device H1.

Same goes with send & no-send options

config node H1 interface eth1 traceoptions flag all

Enable send and recv logging only for interface eth1 of device H1. Logs shall be written in file logs/H1-eth1.txt. There is “→” and “←” mark in log file which shows which pkt is sent or recvd on an interface. If any device level logging is enabled, logs shall continue to be written in H1.txt.

config node H1 interface eth1 traceoptions flag send

Enable only send logging only for interface eth1 of device H1. Logs shall be written in file logs/H1-eth1.txt

config node H1 interface eth1 traceoptions flag no-send

Disable send logging only for interface eth1 of device H1.

Same goes with recv & no-recv options

8.2 Console logging

Note that, sometimes you may want to see the logs on the console rather than log file for quick debugging. For that, you need to enable stdout logging as below.

config global stdout

conf node H1 traceoptions flag stdout

The above two configs shall enable logging for H1 on console itself. Logging in H1.txt shall be uninterrupted.

"config global stdout" is like a main-switch. Disabling it would *temporarily suppress* console logging for entire topology devices and interfaces. Enabling it again restore console logging for all entities (devices or interfaces) for which *stdout* option is individually set. This CLI do not impact logging in log files.

Similarly, you can enable console logging per interface level also using below cli.

config node H1 interface eth1 traceoptions flag stdout

Use "no-stdout" to disable console logging for a device or a particular interface.

Example :

conf node H1 traceoptions flag no-stdout

No device level logs shall be written to console.

config node H1 interface eth1 traceoptions flag no-stdout

No logs related to pkts being sent or recvd on eth1 intf of device H1 shall be written on console provided that 'config node H1 traceoptions flag stdout' is not set. Little confusing !! you will get used to it. :p

8.3 Logging Status

You may want to check what all loggings are enabled on a device and its interfaces. For that use cmd:

show node H1 log-status

Example:

```
tcp-ip-project> $ show node H1 lo
Parse Success.
Log Status : Device : H1
        all     : ON
        recv    : ON
        send    : ON
        stdout  : OFF
        Log Status : eth0/1(UP)
                all     : OFF
                recv    : OFF
                send    : OFF
                stdout  : ON
```

8.4 Trace logs

You don’t have to enable any logging to enable tracelogs. All tracelogs are always written into file logs/tcp_log_file. Tracelogs are internal printf which are written into this log file. This log file do show any packet, but show only internally placed printfs. This log gile would contain logs or all devices in the topology.

Example:

isis_schedule_lsp_pkt_generation (332) :R0: ISIS(LSPDB MGMT) : LSP generation scheduled, reason : ISIS EVENT PROTOCOL ENABLED

isis_generate_lsp_pkt (300) :R0: ISIS(LSPDB MGMT) : Self-LSP Generation task triggered

isis_install_lsp (106) :R0: ISIS(LSPDB MGMT) : Lsp Recvd : 122.1.1.0-1(0x7f81900033b1) on intf (null), old lsp : (null)-0((nil))

isis_install_lsp (132) :R0: ISIS(LSPDB MGMT) : Event : ISIS EVENT SELF FRESH LSP

isis_install_lsp (149) :R0: ISIS(LSPDB MGMT) : Event : ISIS EVENT SELF FRESH LSP : LSP to be Added in LSPDB and flood

isis_add_lsp_pkt_in_lspdb (772) :R0: ISIS(LSPDB MGMT) : LSP 122.1.1.0-1 added to lspdb

Notice the format of each line above. Logs are printed in following format :

<fn name> (<line no>) : <device name>: <feature name> : <user defined log string>

Feel free to grep the contents of this file based on device name to see the logs related to one device only.

8.5 Ping

8.6 Supported show commands

9 ADDING CUSTOM CLIs

It is evident, if you are writing a new application protocol, you would need a way to configure and interact with the application protocol. The TCP/IP stack library provides a mechanism to allow you to add new additional custom CLIs using which you can control and configure user written application protocol or TCP/IP stack.

9.1 Built-In CLIs

TCP/IP Stack already comes with the variety of inbuilt commands listed below. You must try out some commands to get familiar with the CLI interface of TCP/IP library.

vm@ubuntu:~/Documents/tcpip_stack$ ./tcpstack.exe

run - 'show help' cmd to learn more

tcp-ip-project> $. (<< Just type . <dot> to list all supported commands)

Parse Success.

ROOT show help

(show how to use this CLI interface)

ROOT show history <N>

(show last N commands triggered)

ROOT show history

(show history of all commands triggered)

ROOT show registered commands

(show all supported commands)

ROOT show topology node <node-name>

(show topology details of a particular node)

ROOT show topology

(show complete topology details)

ROOT show node <node-name> log-status

(show debugging status of a particular device/node of a topology)

ROOT show node <node-name> ddcp-db

(DDCP protocol specific: show ddcp database on a node)

ROOT show node <node-name> spf-result

(show shortest path first result computed by a node)

ROOT show node <node-name> arp

(show ARP table of a node)

ROOT show node <node-name> mac

(show MAC table of a node)

ROOT show node <node-name> rt

(show routing table of a node)

ROOT show node <node-name> interface statistics

(show interface statistics (send & recv pkts) of all interfaces on a node)

ROOT debug show cmdtree

(dump command tree, used to debug when you are adding new CLIs)

ROOT debug show node <node-name> timer

(show timer Data structure of a node)

ROOT config supportsave enable

(Not functional)

ROOT config console name <cons-name>

(change the console name of the CLI prompt)

ROOT config global stdout

(Enable logging on standard output)

ROOT config global no-stdout

(Disable logging on standard output)

ROOT config node <node-name> traceoptions flag <flag-val>

(Enable logging for a particular node)

ROOT config node <node-name> interface <if-name> traceoptions flag <flag-val>

(Enable logging for a particular interface on a given node)

ROOT config node <node-name> interface <if-name> l2mode <l2-mode-val>

(Configure L2 interface in ACCESS or TRUNK Mode)

ROOT config node <node-name> interface <if-name> <if-up-down>

(Enable or Disable the interface)

ROOT config node <node-name> interface <if-name> vlan <vlan-id>

(Configure Vlans on an interface)

ROOT config node <node-name> route <ip-address> <mask> <gw-ip> <oif>

(Install the L3 Route on a Node (L3 Router))

ROOT config node <node-name> route <ip-address> <mask>

(Install the L3 Route on a Node (L3 Router) without OIF (Direct Route), not used)

ROOT config

(Enter Config Mode)

ROOT run spf all

(Run SPF on all L3 nodes of a topology. In other words update the L3 routing table on all L3 routers of a topology)

ROOT run node <node-name> ping <ip-address> ero <ero-ip-address>

(Ping using ERO)

ROOT run node <node-name> ping <ip-address>

(Ping an IP-address and test reachability)

ROOT run node <node-name> ddcp-query periodic <ddcp-q-interval>

(DDCP Protocol specific : Fire DDCP periodic Queries)

ROOT run node <node-name> ddcp-query

(DDCP Protocol specific : Fire one DDCP periodic Query)

ROOT run node <node-name> resolve-arp <ip-address>

(Manually Resolve ARP for a IP-address)

ROOT run node <node-name> spf

(Refresh L3 touting table on a particular node)

ROOT repeat

(Re-trigger the last Command)

ROOT cls

(Clear the Console Screen)

ROOT cd..

(Go one level up in cmd tree)

ROOT cd

(Go to Top of cmd tree)

You can see, any command starts with either config, show, run, debug or clear. Some miscellaneous commands are also provided such as *repeat cls cd.. cd .* Try out these commands to know what they do.

9.2 Extra CLI functionality for Convenience

?

Shows the next set of possible expected keywords

/ <enter>

Enter into a particular level in command hierarchy

do

run show commands from within config hierarchy directly.

Example:

tcp-ip-project> $ conf node /

Parse Success.

tcp-ip-project> config-node $ do show node R1 rt

Parse Success.

9.3 Adding a New Customized CLI

10 Traffic Generation

11 UTILITIES

This Section list the common APIs/structures to be used to work with TCP/IP stack library.

11.1 (De)Malloc-ing a new Packet

In your TCP/IP application, it is quite obvious that you would want to prepare and free network packets. To prepare a new packet, you would want a pkt buffer memory. Once you know the size of the pkt you want to prepare, You can use below API to request memory allocation for the pkt.

char *tcp_ip_get_new_pkt_buffer (uint32_t pkt_size);

The above API returns the buffer pointer which can be used to store pkt content. Once you get the pointer to the pkt buffer from above API, you can modify the contents of the pkt as per the problem statement.

Once you are done with the pkt, you can release the pkt memory using below API :

void tcp_ip_free_pkt_buffer (char *pkt, uint32_t pkt_size) ;

Note: Don't use Direct calloc/malloc standard functions for memory allocations for packets.

11.2 Ip Address Conversions

tcpip_stack/utils.h tcpip_stack/utils.c file provide some utility APIs to be used. You must take a look at these files and see what there for you to use.

11.2.1 char * tcp_ip_covert_ip_n_to_p (uint32_t ip_addr, char *output_buffer);

Convert an IP address from uint32 format into A.B.C.D format. Either Caller supply its own output buffer (2nd arg) otherwise Library will return its internal static memory buffer with the output.

11.2.2 uint32_t tcp_ip_covert_ip_p_to_n (char *ip_addr)

Convert an IP address from A.B.C.D format into uint32 format.

11.2.3 Iterate Over TLV Buffer

TCP/IP Stack Library provides a handy Iterator macro to allow you to loop over TLV buffer with ease.

The macros are :

#define ITERATE_TLV_BEGIN(start_ptr, type, length, tlv_ptr, tlv_size)

#define ITERATE_TLV_END(start_ptr, type, length, tlv_ptr, tlv_size)

Above macros can be found in *utils.h* file.

12 WORKING WITH TIMERS

Timers are one of the most commonly used data structures in Networking. In Networking it is very common scenario to schedule the events to trigger in future. For example, sending some pkt after 30 sec, Or sending some pkt after every 30 sec Or deleting some data structure if Event E do not happen within 15 sec from now etc.

TCP/IP Stack is already integrated with Timer library to be used. The granularity of this timer library is 1 sec. Timer Library code is in **tcpip_stack/ WheelTimer**.

When Topology is loaded, every Device(Node) has its own Timer instance running. The pointer to this Timer instance can be obtained as:

```
wheel_timer_t *wt = GET_NODE_TIMER_INSTANCE (node_t* node);
```

When application schedule an event with the timer, timer library returns the handle of type *wheel_timer_elem_t* to the application for the event. Using this handle, application can re-schedule, cancel or perform other operation on the event.

12.1 How to Schedule a Future Event

Here Event could be anything – sending a periodic pkt, deletion of some data structure in future etc. Application need to invoke the below API to schedule the future event.

```
wheel_timer_elem_t *
register_app_event(wheel_timer_t *wt,          /*Device's Timer instance*/
                   app_call_back call_back, /*function pointer, the actual event to trigger*/
                   void *arg,                  /*Argument to fn pointer*/
                   int arg_size,               /*Arg Size in Bytes*/
                   int time_interval,          /*Time interval after which Event is to be trigger*/
                   char is_recursive);       /*1 – if the event is to be trigger repeatedly after every
                                               'time_interval' sec. 0 if only once*/
```

app_call_back call_back is a pointer to the function which application must provide as a 2nd argument. Fn prototype should be

```
void <fn-name>(void *, int );
```

This fn shall be invoked by the Timer library as per the 'time interval' and 'is_recursive' parameters provided.

example of one such function is :

```
void transmit_hellos(void *arg, int sizeof_arg);
```

Example of Creating a timer Event :

```
wheel_timer_elem_t *wt_elem =
register_app_event(GET_NODE_TIMER_INSTANCE(interface->att_node,
                                           transmit_hellos,
```

```
                                        (void *)&pkt_meta_data,
                                        sizeof(pkt_meta_data_t),
                                        interval_sec,
                                        is_repeat ? 1 : 0);
```

Note :

1. The argument *arg* (3rd argument) is the Application's data structure on which the callback function (2nd argument) will act upon. Once the event is registered, the application data structure and its size can be obtained from handle element returned as below:

```
wt_elem->arg;
wt_elem-> arg_size;
```

12.2 Cancelling the Scheduled future event

```
void de_register_app_event(wheel_timer_t *wt,
                           wheel_timer_elem_t *wt_elem);
```

12.3 To Re-schedule the already Scheduled Event

```
void
wt_elem_reschedule(wheel_timer_t *wt,
                   wheel_timer_elem_t *wt_elem,
                   int new_time_interval);
```

12.4 Getting the Remaining time left

```
int
wt_get_remaining_time(wheel_timer_t *wt,
                      wheel_timer_elem_t *wt_elem);
```

13 Memory Management

14 Additional Libraries

14.1 LinkedList Library

14.2 Tree Library

14.3 Timer Library

14.4 Serialized Buffer Library

14.5 BitOp

15 PROJECTS BUILT ON TCP/IP STACK LIBRARY

15.1 Neighborship Management

15.2 DDCP (Distributed Data Collection Protocol)

16 CAVEATS

As of Today (14 July 2020), TCP/IP stack library do not support Inter-Vlan routing.

17 CONTACT

To report any Bug in library, pls write to

Email: sachinites@gmail.com

Visit Website: www.csepracticals.com for more projects and courses.

18 CASE STUDY: NETWORK APPLICATION PROTOCOL IMPLEMENTATION EXAMPLE

This Section presents the Case study in which we implement the Pseudo Interior Gateway Protocol called **IS-IS** (**Intermediate System to Intermediate System**). You may be unfamiliar with the name; you can think of this protocol as twin sister of more popular similar protocol **OSPF (Open Shortest Path First)** Protocol. Both ISIS and OSPF are IGP protocols and fall under the category of **link-state protocols** and achieves the same objectives.

We shall implement the Simplified version of actual ISIS protocol using our tcp-ip stack library. The Goal is to learn the nuances involved in implementing a typical Network protocol on a device from absolute scratch. We shall not be complying to the standards and relax or simplify certain aspects of ISIS protocol to prevent over-complicate the project. A typical Network protocol, be it ISIS or any other, involves a lot of complexity and implementing it complete from scratch while complying with the standards shall be over-kill over us. You will be doing it all your life once you Join the industry, but here, this case study aims at giving you firsthand experience regarding what it takes to implement a typical Network protocol on a (simulated) device.

After doing this Course, you can proudly make a mention in your resume as follow:

Implemented Pseudo IGP protocol (= simplified ISIS as an example) from Scratch in C. The following Goals of the project were achieved:

1. *Implemented Config/Show/Debug CLIs – Front end with backend handlers.*
2. *Parsing and Cooking of Protocol Packets (Hellos, Link-state-Packets)*
3. *Implementing Neighborship Management State Machine*
4. *Building and Advertising Link State Database*
5. *Implemented Dijkstra Algorithm to Compute Routes*
6. *Route Installation in Routing Information Base (RIB)*
7. *Protocol Reaction towards external events such as Link down, Up, nbrship down, up etc*

Project Limitation:

Link-State DB synchronization State machine was not implemented in s simplified way (not as per RFC)

It's a simplified implementation, no L1 L2 Level hierarchy was implemented.

The implementation was implemented on Topology Simulator software library, not eligible for any production deployment.

I bet, if you are giving an interview in the company for a *network developer role*, entry level up to 2-3 yrs of experience or making a cross-domain switch into Networking Development, seeing the above mention of the project in your resume would give goosebumps to the interviewer. If you could answer cross–questions (I shall be covering in this Course) – you shall be selected without any doubt. Maintain github to present your codes to him if asked. It is important to mention the limitation of the project before hand to prevent the interviewer from asking questions from the area you are not aware of. In this case we mentioned that LSPDB synchronization was not implemented. The reason is – It's too complex for you at this level (Entry level to 2-3 yrs of experience). I don't want to over-kill you guys with too much complexity that could kill your interest in this project/course.

18.1 Project Goals

The AIM of this project is to cycle you through the experience of end-to-end implementation of a typical network protocol. In this case study we have chosen a routing protocol as an example, but the high-level logistics involved to implement a typical network protocol is more or less the same. For example, a typical network protocol has to:

1. Configurable via CLIs
2. Show internal states and results through show CLIs
3. Respond to generic configuration changes (such as link shut-down, IP Address on interface change etc)
4. Compute results and install the results in Tables (Routing Information Base, MAC Tables, hardware Tables etc)
5. Respond to Topological Changes (link failures, device failures etc)
6. Time-out stale Data structures if any.
7. Periodically Generate Or Process Protocol packet
8. Interaction with other module (either information consumer or producer)
9. How to add an additional feature to existing working Protocol Code base. (This is what you shall be doing all your life as a software engineer !!)

Since the project is quite big (I am expecting around 15k LOCs), ***you would also get the opportunity to learn how to :***

1. Organize the code in header and src files

2. Modularize the code base : How to keep the code of different features in different src files
3. Testing the new feature, and cross check it doesn't break existing features
4. Maintain Code Commits through Version control system (github in this case)
5. Bug Fixing, and exercise various debugging techniques (gdb, valgrind etc)

Needless to say, you Can't exercise above points unless you do a project of considerable size.

18.2 Pre-requisites to do this Course

Needless to say, this project is very challenging, and you need to be reasonably good at Data Structures and Algorithms in C/C++ programming. If you are still struggling with linkedlist/Trees or other common Computer Science basic fundamentals, I would not recommend you spend time on this project – rather work on to build your basics first. There is no point in show casing this project on your resume while at the same time you cannot answer other fundamentals questions such as on Heap Memory, Multithreading etc or fail miserably in reversing a linked list.

Also, I shall assume that you have no background in Networking routing protocol (but do possess Networking Basics such as L3 routing etc) therefore I shall begin from absolute scratch, covering all necessary theories before we start coding. Along the journey in the implementation, we shall pick up various new concepts related to Networking, Project Planning, feature designing, or Coding Standards at the Industry level.