**Tools & Technologies used:**

**Programming Language:** C/C++

**Compiler:** GCC

**Reference Packages:** Quagga Routing Suite

# Reference Documents: A Border Gateway Protocol 4 (BGP-4) RFC (<https://tools.ietf.org/html/rfc4271#section-4.3>),

# <https://www.informit.com/library/content.aspx?b=CCIE_Practical_Studies_II&seqNum=77>

# Path Vector Protocol, Book Network Routing, Algorithms protocols and Architectures, Dr.Deep Medhi

**Platform:** Linux Desktop Machine (Lubuntu)

**Others:** Make,Cscope**,** Wireshark**,** VIM

**Protocol Summary:**

**(Reference Wiki):**

Path vector protocol is a network [routing protocol](https://en.wikipedia.org/wiki/Routing_protocol) maintains the path information that gets updated dynamically. The main advantage is that the updates which have looped through the network and returned to the same node are easily detected and discarded. This algorithm is sometimes used in [Bellman–Ford routing algorithms](https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm) to avoid "Count to Infinity" problems.

It is different from the [distance vector routing](https://en.wikipedia.org/wiki/Distance_vector_routing) and [link state routing](https://en.wikipedia.org/wiki/Link_state_routing). Each entry in the routing table contains the destination network, the next router and the path to reach the destination.

Path Vector Messages in [Border Gateway Protocol](https://en.wikipedia.org/wiki/Border_Gateway_Protocol) (BGP): The [autonomous system boundary routers](https://en.wikipedia.org/wiki/Open_Shortest_Path_First#Autonomous_system_boundary_router) (ASBR), which participate in path vector routing, advertise the reachability of networks. Each router that receives a path vector message must verify that the advertised path is according to its policy. If the messages comply with the policy, the ASBR modifies its routing table and the message before sending it to the next neighbor. In the modified message it sends its own AS number and replaces the next router entry with its own identification.

Border Gateway Protocol is an example of a path vector protocol. In BGP, the routing table maintains the [autonomous systems](https://en.wikipedia.org/wiki/Autonomous_system_(Internet)) that are traversed in order to reach the destination system. [Exterior Gateway Protocol](https://en.wikipedia.org/wiki/Exterior_Gateway_Protocol) (EGP) does not use path vectors.

It has three phases:

1. Initiation
2. Sharing
3. Updating

**Protocol Design:**

We have implemented a minimal path vector protocol on TCP using ring topology. Protocol consists of a single update message which packs all the necessary fields’ information into one message and updates to neighbors.

The Protocol is BGP like. Each instance of the implementation acts as BGP router and an AS. We have simulated 5 routers in the design as a ring topology.

In Linux, alias interfaces are created for assigning unique IP to each interface so the each router instance of execution can be assumed to be separate network.

Pay load of the path vector is in TCP. Its entire bit structure is shown below.

We have defined the attributes, which will exchange the path vector information with its neighbors. They are encoded in the TLV format. Fields which are updated are cost, AS fields, next hot and NLRI fields.

**Protocol Setup:**

**IP’s and Listening Ports:**

We used 5 node topology. Each node acts like Router which sends and receives path vector protocol messages.

**5 Autonomous System IP’s:**

#define AS1\_IP "1.1.1.1"

#define AS2\_IP "2.2.2.2"

#define AS3\_IP "3.3.3.3"

#define AS4\_IP "4.4.4.4"

#define AS5\_IP "5.5.5.5"

**5 Autonomous System TCP Listening Ports:**

#define AS1\_PORT 55552

#define AS2\_PORT 55553

#define AS3\_PORT 55554

#define AS4\_PORT 55555

#define AS5\_PORT 55556

**Alias Interfaces Created on Linux:**

sudo ifconfig eth0:1 1.1.1.1 netmask 255.255.255.0 up

sudo ifconfig eth0:2 2.2.2.2 netmask 255.255.255.0 up

sudo ifconfig eth0:3 3.3.3.3 netmask 255.255.255.0 up

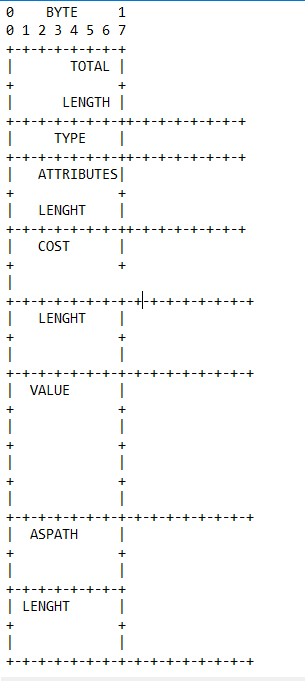
sudo ifconfig eth0:4 4.4.4.4 netmask 255.255.255.0 up

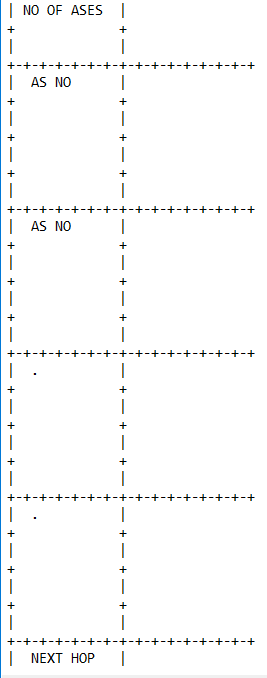
sudo ifconfig eth0:5 5.5.5.5 netmask 255.255.255.0 up

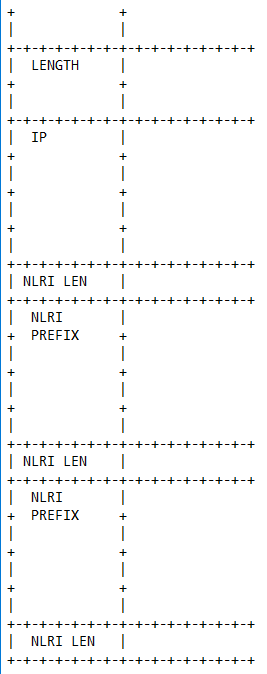
**Packet Format:**

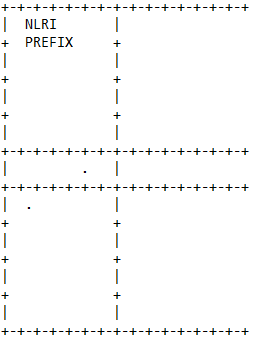
Payload of the TCP contains actual Path vector details i.e Path vector Header and data

Ddd









Here we have used, **Message Type** is path vector = 0x02

**Attributes Type can be**

COST (Type Code 0): 4 bytes fixed

AS\_PATH (Type Code 1): Variable bytes

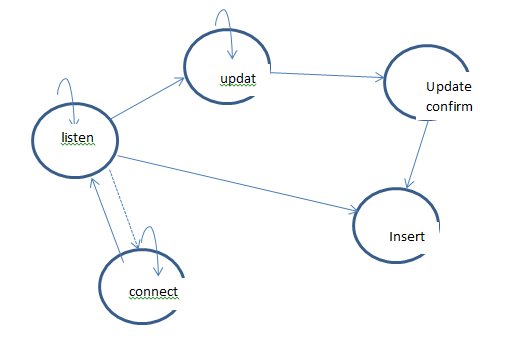
LENGTH: number of ASes. 2 Bytes Fixed.

VALUE: one or more AS Numbers. Each of 32 Bytes

NEXT\_HOP (Type Code 2): 4 Bytes Fixed

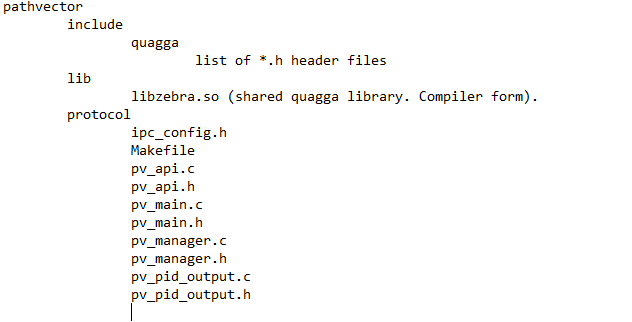
Packet format is bit oriented. Minimum size of the payload is 36 if it is filled with one IP, AS number and one cost.

**State Machine:**



**Code Overview:**

We took reference from the Quagga libraries. We used some needed libraries which make coding easier and efficient. We also looked into BGP protocol code in the Quagga suite to understand flow.



Pv\_manager.c 🡪 In this file real state machine of the code is implemented.

Pv\_manager.h 🡪 Contains protocol related data structures.

Pv\_main.h 🡪 Contains General Data Structures.

Pv\_main.c 🡪 Starting point of program execution.

Ipc\_config.c 🡪 Contains Linux permission handling routines.

Makefile 🡪 Make compilation input.

Pv\_api.c 🡪 Contains useful API’s used in project.

Pv\_api.h 🡪 Contains useful API’s data structures used in project.

Pv\_pid\_output.c 🡪 process PID numbers

**Main Data Structures:**

typedef struct {

pv\_message\_header pv\_message\_header;

uint16\_t path\_attr\_len ; /\*total length of the Path Attributes field in bytes excluding attr len feild\*/

struct list \*path\_attributes\_list;

struct list \*pv\_nlri\_list;

}pv\_message;

typedef struct {

uint16\_t total\_length ; /\* the total length of the message, including the header in bytes\*/

uint8\_t type; /\* 2 - UPDATE \*/

}pv\_message\_header;

typedef struct {

uint16\_t type;

uint16\_t length;

void \*value;

}path\_attributes;

/\*typedef struct {

struct list \*path\_attributes\_list ;

}path\_attributes\_list;\*/

typedef struct {

uint32\_t ases\_number;

} ases\_path\_number;

typedef struct {

uint8\_t no\_of\_ases ;

struct list \*ases\_path\_list;

} ases\_path\_attribute;

/\* NLRI \*/

typedef struct

{

uint8\_t nlri\_length;

uint32\_t nlri\_prefix;

}pv\_nlri;

typedef struct pv\_routing\_table\_row\_s

{

struct route\_node \*route\_node\_row; // containe prefix\_ipv4 info

uint32\_t nexthop\_IP;

uint32\_t cost;

struct list \*ases\_path\_number\_list;

}pv\_routing\_table\_row;

**High level API’s:**

int wr\_pv\_init(int);

int setup\_connections\_to\_neighbour(unsigned char);

int pv\_event(enum pv\_event, int, void \*);

int pv\_listener ();

int pv\_accept (struct thread \*);

int pv\_read\_packet (struct thread \*);

int pv\_cleanup\_pv\_metadata(pv\_message\_metadata \*);

int pv\_parse\_pure\_payload(pv\_message\_metadata \*, void \*, int , int, pv\_peer\_accept\_info \*);

int pv\_delete\_route\_column(struct prefix\_ipv4 \*);

int pv\_get\_route\_cost\_in\_rt\_row(struct prefix\_ipv4 \*, int \*);

int pv\_add\_route\_to\_rt\_row(pv\_message\_metadata \*, int ) ;

int pv\_check\_loop(pv\_message\_metadata \*);

int pv\_open\_connection\_to\_neighbours (struct thread \*);

int pv\_open\_connection(struct sockaddr\_in , int \*);

int send\_advt\_to\_neighbours(pv\_message\_metadata \*, int );

int pkt\_out\_on\_wire(pv\_message\_metadata \*, int , int );

int pv\_build\_self\_metadata();

int pv\_adv\_self\_prefix();

int pv\_print\_route\_table\_to\_file(struct thread \*);

int pv\_check\_route\_in\_rt\_column(struct prefix\_ipv4 \*, struct route\_node \*\*);

int pv\_add\_route\_column(struct prefix\_ipv4 \*p, struct route\_node \*\*out);

int pv\_print\_route\_table\_to\_file\_new();

**The following are the states from implementation point of view.**

**PV\_LISTENER\_READ** -- It will listen on TCP listen socket file descriptor.

**PV\_ACCEPT\_PEER\_READ\_MESSAGE** -- It will listen on TCP Accept socket file descriptor.

**PV\_CLIENT\_CONNECT\_EVENT** – It will try to connect to neighbors and maintain connection.

**PV\_CLIENT\_CONNECT\_TIMER** – Periodically for every 2 sec, it connect to clients if in the first case connection fails.

**PV\_MY\_UPDATE\_TIMER** – It updates routes to neighbors periodically.

**PV\_PRINT\_RT\_TABLE\_TIMER** – It will print route table to file for every 2 sec.

**Compilation:**

Just do

**Sudo make clean**

**Sudo make**

Ignore all warnings.

For running,

**Sudo ./test 1 > 1.txt**

**Sudo ./test 2 > 2.txt**

**Sudo ./test 3 >3.txt**

**Sudo ./test 4 >4.txt**

**Sudo ./test 5 >5.txt**

As libzebra.so is supplied in the code, no need to generate it.

We compiled on Linux Ubuntu version 14.04. For others please compile Quagga, extract library libzebra.so and replace it in the relevant position so that linking errors can be avoided.

When the program compiled successfully, executable file **test** is produced.

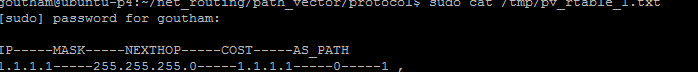
**Demo:**

Open five terminals after creating alias interfaces.

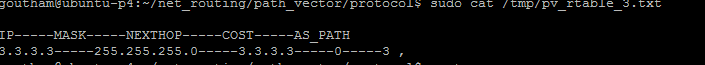
Run the test in each program by specifying command line argument, a number, which signify as a AS number to router and a separate log file name for each router.

To check the logs just open any log file in editor and the protocol packet exchanges can be seen.

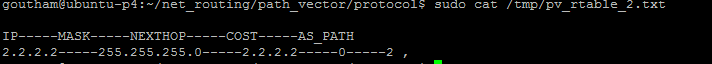
**Sample Routing Table of 1 AS Router:**



**Sample Routing Table of 3 AS Router:**



**Sample Routing Table of 2 AS Router:**



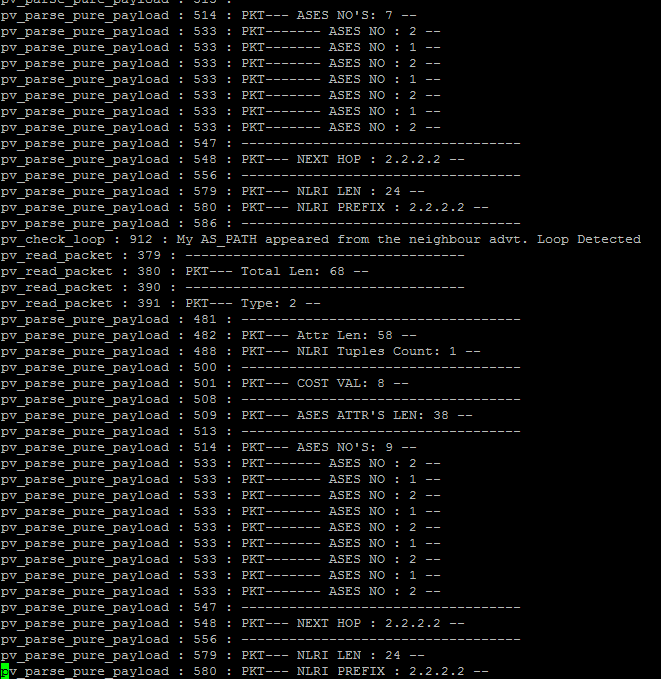
**Sample Routing Table of 4 AS Router:**



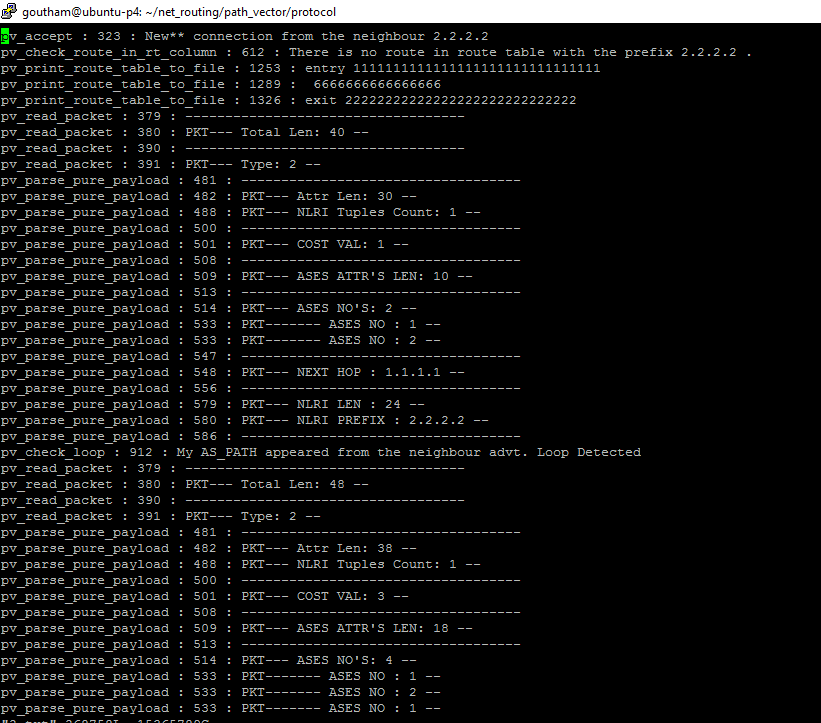
**Sample Routing Table of 5 AS Router:**



**Log file of Router 1:**



**Log file of Router 2:**



From the log files, we can infer that, initially all Routers will try to advertise its neighbor information to its remaining neighbors.

When the neighbors get information, it will parse them and add the next hop field as its IP, increment cost and advertise them to the remaining neighbor. In this case it will not advertise to the same neighbor from which the advertisement is received.

In each advertisement, all the packet level information is filled in the network byte order.

**Issues in Implementation:**

* We did used Quagga routing libraries.
* We have used IP Route table’s data structures, Trie data structures from Quagga. We have seen some wrong pointer references to these tables when trying to scan route tables.
* Route tables are not properly updated because of the above mentioned issues.

**Future Work:**

* We need to add policy implementation, dynamically advertise routes, update routing tables properly, plan to make code work on Docker VM(In this case each Docker VM acts as an router, independent of each other), insert route information to Linux kernel routing table.
* Also we can provide the option to specify IP’s option to users so that they can specify their own AS number IP; s.
* Viewing route tables can be made using some telnet access to each router process instead of updating to file manually.