IP over UDP README

Author: Shan Lu(slu5), Jingyiping Zhang(jzhang12)

For better reading experience on this document, please click here to oepn a rich-formatted PDF file.

Data Structures

```
type RouteInfo struct {
    destIp  string
    cost    uint8
    nextHop string
    ttl     uint8
}
```

```
type InterfaceInfo struct {
    link        *LinkLayer
    active      bool
    phyAddrFrom string
    phyAddrTo   string
    virIpFrom   *net.IPAddr
    virIpTo     *net.IPAddr
    id          int
}
```

```
type AppLayer struct {
    network *NetworkLayer
}
```

```
type NetworkLayer struct {
    registerTable map[uint8](func([]byte, uint32, string))
    routings      map[uint32]*RouteInfo
    app           *AppLayer
    link          *LinkLayer
    phyAddr       *string
    lock          sync.Mutex
}
```

```
type LinkLayer struct {
    network    *NetworkLayer
    conn       *net.UDPConn
    interfaces []InterfaceInfo
}
```

```
type NodeInfo struct {
    phyAddr string
    app     *AppLayer
    network *NetworkLayer
    link    *LinkLayer
}
```

In this project, we designed 6 custom structures to abstract and represent data regarding routing table, link layer interfaces, node and 3-layer network hierachy.

Firstly, `NodeInfo` contains its physical address and pointers to all three layers. Each layer in the network hierachy contains pointers only to its neighbors. Concretely, it means `AppLayer` can only get access to `NetworkLayer`, and `NetworkLayer` holds pointers to both `AppLayer` and `LinkLayer`.

Within link layer, we store an array of interfaces(of type `InterfaceInfo`) and a UDP connection to other “physically-linked” nodes.

At network layer, we have a routing table(represented by a uint32 to `RouteInfo` map), a mutex lock and registerTable. The routing table takes destination's virtual IP address as its key and destIp, cost, nextHop, ttl as its value tuple. This design follows the RIPv2 protocol specification. When routing IP packet, each node unwraps the packet to retrieve its destination, checks the routing table, and sends the wrapped IP packet to the next node that `nextHop` implies. `cost` is a non-negative value associated with each route ranging from 0 to 16. Specifically, 16 represents a route with infinity cost. According to RIPv2 protocol, when node closes its interface, it should send a cost of 16 to its closing to all connected neighbor nodes. This triggered update is helpful to detect some "count to infinity" problem. Time to live, abbriviated as `ttl`, tells node the correctness of this routing information is only guaranteed within this amount of time. Whenever some routes time out, it will be removed from the routing table. In our implementation, each node sends periodical broadcast message of its routing table every 5 seconds, and one route information times out when this node fails to get this routing information again in the next 12 seconds. `registerTable` is a map from `uint8` to a function pointer of prototype `([]byte, uint32, string)`. This allows registering any application layer handler at network layer. The key will be extracted from IP packet header's protocol slot, and all the payload data will be sent directly to the mapped handler as its first argument. The second argument is source IP address, and the third string is in fact the physical address that this packet was sent through. Currently we registered `RIPhandler`(200), `printService`(0) and `reversePrint`(188). IP packets with other protocol number will be simply printed out using printService.

Although RIP handler lives at application layer, sometime this network application has to visit interfaces that live at link layer. For example, when RIP is about to send triggered updates, it has to iterate all interfaces and send RIP entries with respect to each neighboring node. What's more, in poison reverse implementation, it has to know where it sends so that a cost of 16 to the destination node's virtual IP address could be set accordingly.

Installation

To install and run nodes, you have to move src/node and src/nodeUtil into $GOPATH. Then issue the following command:

```
$ mv src/node $GOPATH
$ mv src/nodeUtil $GOPATH
$ cd $GOPATH
$ go install node
```

To run the node, simply give it an `.lnx` file as input argument like:

```
$ ./bin/node A.lnx
```