YOUR FIRST ROUTER

It is essential for the work with the eEx Network Library to understand how the whole Library works.
The Network Library mainly consists of classes which derive from TrafficHandler and classes which derive from Frame.

UNDERSTANDING FRAMES

The Frame classes are a main component of the network library. All implemented network protocols derive from the Frame super class which includes several methods which must be implemented. A frame can also contain a encapsulated frame. When sending a frame, for example, the FrameBytes property is called, which converts the actual frames to bytes and calls the FrameBytes method for the encapsulated frame which again calls the FrameBytes method for the next encapsulated frame and so on.
If you want to implement your own network protocol one day for usage with the eEx Network Library, you will have to implement it as Frame with the frame type UserDefined.
All frame types already contained in the eEx Network Library have got a constructor which accepts a byte array, which will be parsed. Also Ethernet and IP frames automatically create encapsulated frames like ARP, OSPF, TCP or UDP frames.

UNDERSTANDING TRAFFIC HANDLERS

Traffic handlers are like modules in a network application. Traffic handlers have got several **in-** or **output ports** which can be linked to other traffic handlers. Any traffic modifier will accept traffic on the input ports and will forward then handled frames to his output port while any traffic analyzer will accept traffic on the input ports but will not forward the frames to any output handlers.

Because traffic handlers are highly multi-threaded, it is very important to let only handle one traffic analyzer the same frame simultaneously. Basically, this behavior is forced by the architecture network library, but if you want to add analyzers, you should make use of the TrafficSplitter class which clones each frame for the assigned analyzers to avoid concurrent updates or reading.

Someone could say that a application based on the eEx Network Library works like very big graph of moduls where network frames and datagrams travel through.

STARTING YOUR FIRST ROUTER APPLICATION

A router is a very basic component for all networks. It reads frames from its network interfaces, looks up the destination in its routing table, and forwards the frame to the according interface.

Let's start to write a very simple router without user interface, to do so, create a new command line application and add the eEx Network Library to the references.

The first step for building a router is to query all interfaces of the local host. We will use WinPcap interfaces, so WinPcap must be installed on the computer to run the program. At first, we will get all available WinPcapInterfaces and save them into an array:

```
WinPcapInterface[] arWpc = EthernetInterface.GetAllPcapInterfaces();
```

The next step is to create some needed classes, the router, a class which builds ARP tables and so on:

```
Router rRouter = new Router();
TrafficSplitter tsSplitter = new TrafficSplitter();
```

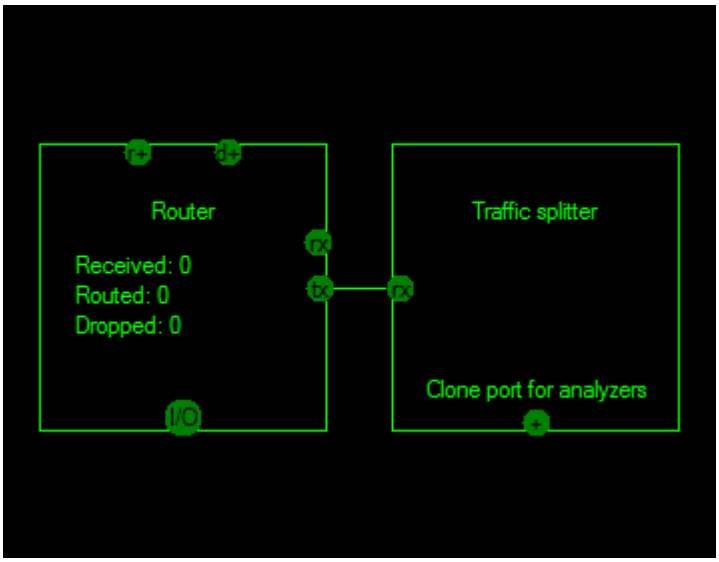

This code snippet creates a Router, which will do your routing and a TrafficSplitter. We will need the traffic splitter later to add some analyzers. The next step is to start this classes so that their worker threads are running and they become ready for handling traffic. Also the handlers must be linked together.

```
//Start handlers
rRouter.Start();
tsSplitter.Start();

//Let the router forward traffic from the interfaces to the traffic
splitter
rRouter.OutputHandler = tsSplitter;
//Let the traffic splitter forward received traffic back to the router
tsSplitter.OutputHandler = rRouter;
```

This code segment starts all handlers. Then it linkes all traffic handlers together in the right way. The linking of handlers does affect the whole behaviour of this program. In this scenario, the graph which is created is not very complicated: The router sends all traffic from the interfaces to the traffic splitter, and the traffic splitter sends it back to the router which will forward it to the interfaces.

Graphically representet, we have got the following scenario now:



The next task we have to do is to add a default route, if we need one. To do so, let's create a new routing entry and simply add it to the router.

```
//Create the properties of the routing entry
IPAddress ipaDestination = IPAddress.Parse("0.0.0.0");
IPAddress ipaGateway = IPAddress.Parse("192.168.1.1");
Subnetmask smMask = Subnetmask.Parse("0.0.0.0");
int iMetric = 10;

//Create the routing entry

RoutingEntry rEntry = new RoutingEntry(ipaDestination, ipaGateway, iMetric,
smMask, RoutingEntryOwner.UserStatic);
```

This code segment creates the IP addresses and the subnetmask for the routing entry and then creates the routing entry with the given metric. Settinig the owner of self added entries to UserStatic is important when working with routing protocols.

Before we add the IP interfaces to the router, there is still one task to go: Let's add some event handlers to see what the router is doing.

```
//Add some event handlers
rRouter.FrameDropped += new EventHandler(rRouter_FrameDropped);
rRouter.FrameForwarded += new EventHandler(rRouter_FrameForwarded);
rRouter.FrameReceived += new EventHandler(rRouter_FrameReceived);
rRouter.ExceptionThrown += new
TrafficHandler.ExceptionEventHandler(rRouter_ExceptionThrown);
```

```
static void rRouter_FrameForwarded(object sender, EventArgs e)
{
    Console.WriteLine("Frame forwarded!");
}

static void rRouter_FrameDropped(object sender, EventArgs e)
{
    Console.WriteLine("Frame dropped!");
}

static void rRouter_ExceptionThrown(object sender, ExceptionEventArgs args)
{
    Console.WriteLine("Router error: " + args.Exception.Message);
}

static void rRouter_FrameReceived(object sender, EventArgs e)
{
    Console.WriteLine("Frame received!");
}
```

Now we will receive a little bit output every time something happens.

The next step is to add the interfaces. Here we have to create an EthernetInterface traffic handler for each WinPcapInterface description we queried in the beginning, start it, and add it to the router. We also have to save the interfaces in a list because we need to close them when we shut down the programm.

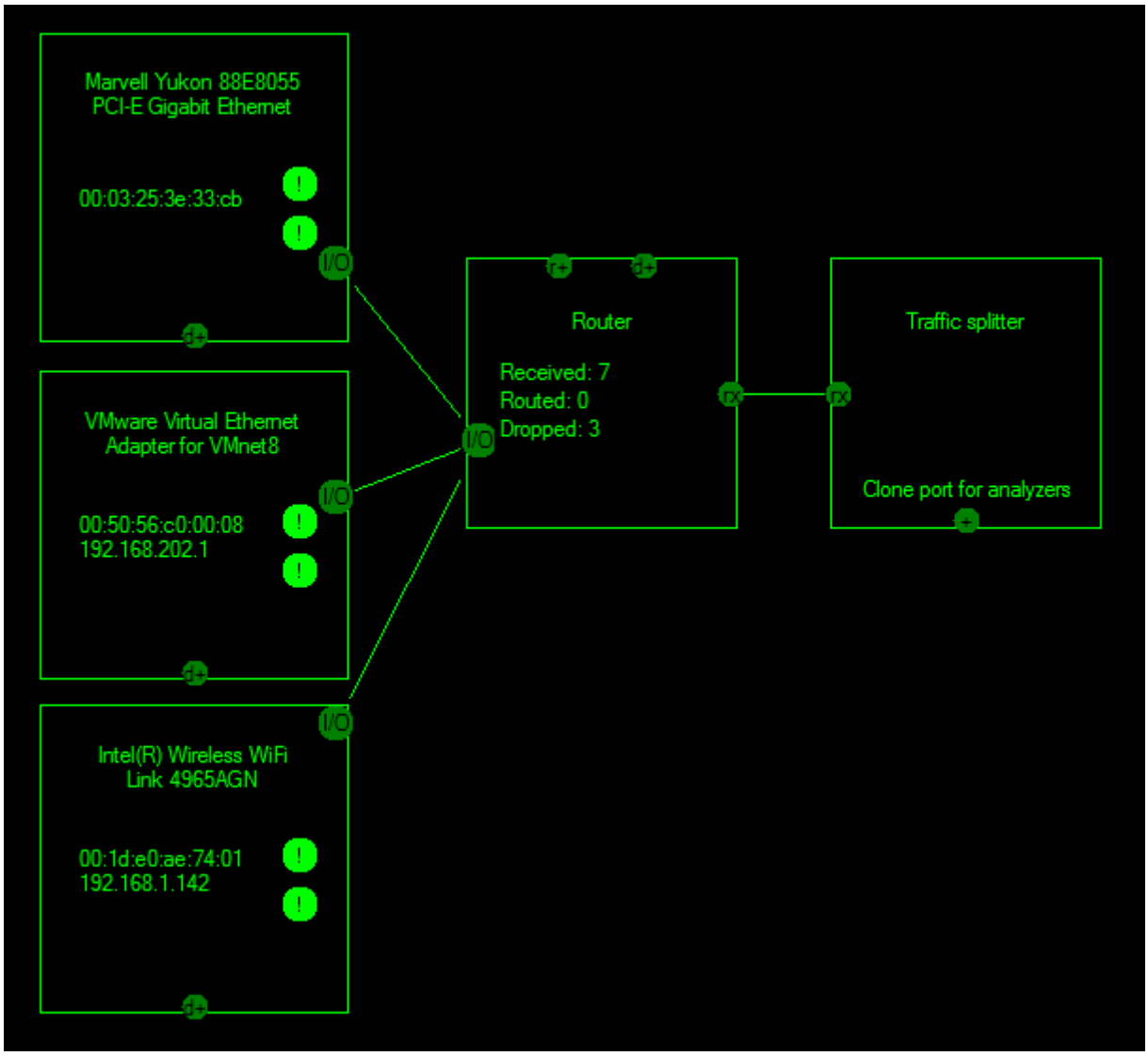
```
//Create a list for the interfaces
List<EthernetInterface> wpcInterfaces = new List<EthernetInterface>();

//Foreach WinPcapInterface of this host
foreach (WinPcapInterface wpc in arWpc)
{
    //Create a new interface handler and start it
    EthernetInterface ipInterface = new EthernetInterface(wpc);
    ipInterface.Start();

    //Then add it to the router and to our list
    wpcInterfaces.Add(ipInterface);
    rRouter.AddInterface(ipInterface);
}
```

Here we creates a new interface handler for each WinPcapInterface and added all known interfaces to the router. The router now inserts routes to the subnets which belong to the interfaces automtically in its routing tables.

Again, our scenario expanded:



If you would start this programm now, it would do routing and it would also generate some output. But if you would close it now, there could be errors because none of the interfaces would be closed and none of the threads would be stopped.

Stopping graphs is sometimes complicated. You have to keep one rule in mind. Traffic handlers should be stopped in the same order as they were started and interfaces should be stopped last. Also you have to call the cleanup method in the same order as described above before initiating the shutdown process.

A call to the cleanup method will stop, for example, the interfaces from receiving traffic or lets attacks restore conditions as they were before the attack happened.

So let's implement that our programm will shut down cleanly when we press the 'x' key.

```
//Run until 'x' is pressed
while (Console.ReadKey().Key != ConsoleKey.X) ;

//Start the cleanup process for all handlers
rRouter.Cleanup();
tsSplitter.Cleanup();

//Start the cleanup process for all interfaces
foreach (EthernetInterface ipInterface in wpcInterfaces)
{
```

```
    ipInterface.Cleanup();
}

//Stop all handlers
rRouter.Stop();
tsSplitter.Stop();

//Stop all interfaces
foreach (EthernetInterface ipInterface in wpcInterfaces)
{
    ipInterface.Stop();

}
```

Now it's time to test the router.