TinyNS Tutorial

Miguel Bazdresch

May 9, 2018

Introduction

TinyTS is a very small and simple network simulator. The main purpose of TinyTS is to enable you to implement, test and evaluate your own routing algorithms in a variety of network topologies. Although this can be done in a larger, professional simulator such as NS3, TinyNS has a number of features that allow you to focus on the fundamentals, instead of spending your time learning to use the tool:

1. In its tiniest form, it is only 149 lines long. This means that you can understand the simulator inside and out, which makes it easier to modify it.
2. It is written in Python, which is a very accessible language.
3. It does not use discrete events. While this limits its power and flexibility, it results in a design that is very simple and easy to understand.

To be sure, I believe you should eventually move on to more powerful tools such as NS3, OpNet, etcetera. However, a first network simulation experience with TinyNS can be a solid foundation on which to build on.

The simulator

There are a few core ideas behind the simulator:

1. There are two kinds of nodes: edge nodes and switches. An edge node generates and consumes packets; a switch forwards packets.
2. Edge nodes have only one port; switches can have any number of ports.
3. Nodes are connected by connecting their ports. Ports are bidirectional.
4. The simulation consists of a repetition of these three steps:
   (a) Edge nodes are allowed to receive one packet.
   (b) Switches forward one packet stored in their memory.
   (c) Edge nodes are allowed to transmit a packet.

Let us look at the code that implements these ideas.

Packets

A packet is implemented as follows:

```
class Packet:
    def __init__(self, s_addr, d_addr, payload):
        self.s_addr = s_addr
        self.d_addr = d_addr
        self.payload = payload
```

The meaning of the variables is as follows:

- `s_addr` is the address of the source, the packet originator.
- `d_addr` is the address of the destination.
- `payload` is the actual data carried by the packet.

Edge nodes

An edge node is defined as follows:

```
class EdgeNode:
    def __init__(self, address, d_addr_set, tx_prob):
        self.address = address
        self.d_addr_set = d_addr_set
        self.tx_prob = tx_prob
        self.memory = []
```

The fields have the following meaning:

- `address` is the node's address. Packets meant for this node need to put this address as destination.
- `d_addr_set` is a list of the possible destinations this node may send packets to. The node will choose among this list at random.
- `tx_prob` is the probability that a node actually generates a packet when given the opportunity.
- `memory` The node's buffer (actually a FIFO). Arriving packets will be placed at the end of the buffer.

A node has two functions to help the simulator tell it apart from a switch:

```
def isedge(self):
    return True

def isswitch(self):
    return False
```

The edge node class has a function to transmit a packet. The function is called with the current simulation time for logging purposes. A uniform-distributed random number is generated, and if its value is less than `tx_prob`, then a packet is generated and returned to the simulator. If no packet is transmitted, then the node returns `None`.

```
def transmit(self, time):
    # Determine if a packet will be generated
    if random() < self.tx_prob:
        # Determine destination address
        d_addr = choice(self.d_addr_set)
```

```
        # instantiate packet
        pkt = Packet(self.address, d_addr, 'dummy payload')
        self.txaction(pkt, time)
        return pkt
    else:
        return None
```

The edge nodes also have a function to receive a packet.

```
def receive(self, time):
    if len(self.memory) > 0:
        pkt = self.memory.pop()
        self.rxaction(pkt, time)
```

Note that these last two functions call two other auxiliary functions. `transmit` calls `txaction`, and `receive` calls `rxaction`. This allows you to specify some additional actions that the edge node will take when transmitting and receiving packets. One example is logging (either to the screen or to a file). More advanced networks may require additional actions. Note that these functions do nothing in the minimal TinyNS implementation.

Switch nodes

A switch node never creates its own packets. All it does is forward packets received through one port through a different port. The output port will hopefully be chosen so that the packet gets closer to its destination.

A switch can have any number of ports. However, the switch can only forward a single packet at a time.

The switch uses a routing table to decide how to forward the packets. An initial routing table needs to be provided when instantiating a switch; however, this table need to be optimal or even correct.

A switch node is defined as follows:

```
class SwitchNode:
    # Class initializer
    def __init__(self, ID, ports, rt):
        self.ID = ID
        self.ports = ports
        self.rt = rt
        self.memory = []
```

The fields have the following meaning:

- `ID` is the switch's unique ID.
- `ports` is a list of the switch's ports.
- `rt` is the switch's routing table, implemented as a dictionary.
- `memory` is the node's buffer memory.

Two functions let the simulator tell an edge node apart from a switch:

```
def isedge(self):
    return False

def isswitch(self):
    return True
```

Packets are forwarded with the following function. If there is a packet in the buffer, then the packet is removed from the buffer and its destination address is obtained. The routing table is consulted to find the port the packet should be forwarded to. If the buffer is empty, then a tuple `(None, None)` is returned.

```
def forward(self, time):
    # if there is something in memory, process it
    if len(self.memory) > 0:
        pkt = self.memory.pop()
        # determine output port
        outport = None
        for i in self.rt:
            if i == pkt.d_addr:
                outport = self.rt[i]
        # execute an action
        self.action(pkt, outport)
        # return packet and output port
        return pkt, outport
    else:
        return None, None
```

Note that, just like with the edge nodes, an action function is invoked when a packet is forwarded. In this minimal implementation, that function does nothing.

The simulator control

The simulation executes what is called a "round robin scheduler", which calls each node in order and asks it to perform a function. The key function is called `run`. It first lets edge nodes receive; then, it calls the switches to forward one packet; finally, it lets the edge nodes transmit. This is executed for the desired number of iterations. This function uses the node list and the connection list to know which nodes to call, and in which node's memory to write the packets as they flow through the network.

```
def run(nodelist, connlist, iterations):
    for i in range(iterations):
        # Edge nodes receive
        for n in nodelist.nodelist:
            if n.isedge():
                n.receive(i)
```

```
        # Switch nodes forward
        for n in nodelist.nodelist:
            if n.isswitch():
                pkt, port = n.forward(i)
                if pkt != None:
                    nextnode = findnextnode(connlist, n, port)
        # Edge nodes transmit
        for n in nodelist.nodelist:
            if n.isedge():
                pkt = n.transmit(i)
                if pkt != None:
                    nextnode = findnextnode(connlist, n, '1')
                    nextnode.memory.insert(0,pkt)
                    nextnode.memory.insert(0,pkt)
    print("Done!")
```

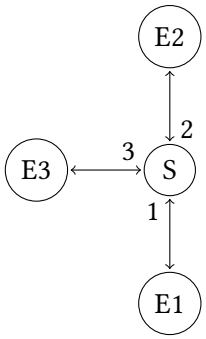
Creating the network

To create a network, you need to instantiate the nodes and connect them. To do that, you'll need a node list and a connection list:

```
# List of connections
class ConnectionList:
    def __init__(self):
        self.connlist = []

    def connect(self, node1, port1, node2, port2):
        self.connlist.append((node1, port1, node2, port2))

# List of nodes
class NodeList:
    def __init__(self):
        self.nodelist = []

    def append_node(self, node):
        self.nodelist.append(node)
```

Putting it all together: an example

Let's say that we want to simulate the following network:



Furthermore, we want each edge node to transmit a different message. We need to follow these steps:

1. Create the node list and connection list.

```
nl = NodeList()
cl = ConnectionList()
```

2. Create the edge nodes and append them to the node list.

```
e1 = EdgeNode('E1',['E2'],1)
e2 = EdgeNode('E2',['E1','E3'],1)
e3 = EdgeNode('E3',['E1','E2'],1)
nl.append_node(e1)
nl.append_node(e2)
nl.append_node(e3)
```

3. Create the switch and append it to the node list.

```
s = SwitchNode('S',['1','2','3'],dict([('E1','1'),('E2','2'),('E3','3')]))
nl.append_node(s)
```

4. Connect the nodes:

```
cl.connect(e1,'1',s,'1')
cl.connect(e2,'1',s,'2')
cl.connect(e3,'1',s,'3')
```

5. Run the simulation for the desired number of steps:

```
run(nl,cl,10)
```

The code to run this simple example can be found in `test/simple.py`.