INET Framework User's Guide

Release 4.7

INET community

Nov 02, 2026

CONTENTS

1 Introduction **3**
- 1.1 What is INET Framework . . . 3
- 1.2 Designed for Experimentation . . . 3
- 1.3 Scope of this Manual . . . 3

2 Using the INET Framework **5**
- 2.1 Installation . . . 5
- 2.2 Installing INET Extensions . . . 5
- 2.3 Getting Familiar with INET . . . 5

3 Networks **7**
- 3.1 Overview . . . 7
- 3.2 Built-in Network Nodes and Other Top-Level Modules . . . 7
- 3.3 Typical Networks . . . 8
- 3.4 Frequent Tasks (How To...) . . . 10

4 Network Nodes **13**
- 4.1 Overview . . . 13
- 4.2 Ingredients . . . 13
- 4.3 Node Architecture . . . 14
- 4.4 Customizing Nodes . . . 14
- 4.5 Custom Network Nodes . . . 15

5 Network Interfaces **17**
- 5.1 Overview . . . 17
- 5.2 Built-in Network Interfaces . . . 17
- 5.3 Anatomy of Network Interfaces . . . 17
- 5.4 The Interface Table . . . 18
- 5.5 Wired Network Interfaces . . . 19
- 5.6 Wireless Network Interfaces . . . 19
- 5.7 Special-Purpose Network Interfaces . . . 20
- 5.8 Custom Network Interfaces . . . 21

6 Applications **23**
- 6.1 Overview . . . 23
- 6.2 TCP applications . . . 23
- 6.3 UDP applications . . . 26
- 6.4 IPv4/IPv6 traffic generators . . . 27
- 6.5 The PingApp application . . . 28
- 6.6 Ethernet applications . . . 28

7 Transport Protocols **29**
- 7.1 Overview . . . 29
- 7.2 TCP . . . 29
- 7.3 UDP . . . 32

7.4 SCTP . . . 32
7.5 RTP . . . 32

8 The IPv4 Protocol Family **33**
8.1 Overview . . . 33
8.2 IPv4 . . . 33
8.3 IPv4 Routing Table . . . 34
8.4 ICMP . . . 34
8.5 ARP . . . 34
8.6 IGMP . . . 35

9 The IPv6 Protocol Family **37**
9.1 Overview . . . 37
9.2 IPv6 . . . 37
9.3 IPv6 Routing Table . . . 38
9.4 ICMPv6 . . . 38
9.5 Neighbor Discovery and Autoconfiguration . . . 38
9.6 IPv6 Tunneling . . . 39
9.7 Mobile IPv6 . . . 39

10 Other Network Protocols **41**
10.1 Overview . . . 41
10.2 Protocols . . . 42
10.3 Address Types . . . 42
10.4 Address Resolution . . . 43

11 Network Autoconfiguration **45**
11.1 Overview . . . 45
11.2 Configuring IPv4 Networks . . . 45
11.3 Configuring IPv6 Networks . . . 53
11.4 Configuring Layer 2 . . . 54
11.5 Configuring Time-Sensitive Networking . . . 54

12 Internet Routing **57**
12.1 Overview . . . 57
12.2 RIP . . . 57
12.3 OSPF . . . 58
12.4 BGP . . . 59

13 Ad Hoc Routing **61**
13.1 Overview . . . 61
13.2 AODV . . . 61
13.3 DSDV . . . 61
13.4 DYMO . . . 62
13.5 GPSR . . . 62

14 Queueing Model **63**
14.1 Overview . . . 63
14.2 Sources . . . 64
14.3 Sinks . . . 64
14.4 Queues . . . 65
14.5 Buffers . . . 65
14.6 Filters . . . 65
14.7 Classifiers . . . 65
14.8 Schedulers . . . 66
14.9 Servers . . . 66
14.10 Markers . . . 66
14.11 Meters . . . 66
14.12 Token generators . . . 67

14.13 Conditioners . . . 67
14.14 Other queueing elements . . . 67

15 Differentiated Services **69**
15.1 Overview . . . 69
15.2 Architecture of NICs . . . 70
15.3 Simple modules . . . 70
15.4 Compound Modules . . . 75

16 The MPLS Models **77**
16.1 Overview . . . 77
16.2 Core Modules . . . 77
16.3 Classifier . . . 80
16.4 MPLS-Enabled Router Models . . . 80

17 Point-to-Point Links **81**
17.1 Overview . . . 81
17.2 The PPP module . . . 81
17.3 PppInterface . . . 82

18 The Ethernet Model **83**
18.1 Overview . . . 83
18.2 Nodes . . . 83
18.3 The Physical Layer . . . 84
18.4 Ethernet Interface . . . 84
18.5 Components . . . 84
18.6 Implemented Standards . . . 86

19 Ethernet 10BASE-T1S **87**
19.1 Overview . . . 87
19.2 Point-to-point links . . . 87
19.3 Multidrop links . . . 87
19.4 Result analysis . . . 88

20 Time-Sensitive Networking **89**
20.1 Devices . . . 89
20.2 Time Synchronization . . . 89
20.3 Per-stream Filtering and Policing . . . 90
20.4 Scheduling and Traffic Shaping . . . 91
20.5 Frame Replication and Elimination . . . 92
20.6 Frame Preemption . . . 93
20.7 Cut-through Switching . . . 94
20.8 Automatic Network Configuration . . . 94

21 The 802.11 Model **97**
21.1 Overview . . . 97
21.2 MAC . . . 97
21.3 Physical Layer . . . 99
21.4 Management . . . 99
21.5 Agent . . . 99

22 The 802.15.4 Model **101**
22.1 Overview . . . 101
22.2 Network Interfaces . . . 101
22.3 Physical Layer . . . 101
22.4 MAC Protocol . . . 102

23 MAC Protocols for Wireless Sensor Networks **103**
23.1 Overview . . . 103
23.2 B-MAC . . . 103

23.3 L-MAC . . . 103
23.4 X-MAC . . . 103

24 Clock Model **105**
24.1 Overview . . . 105
24.2 Clocks . . . 105
24.3 Clock Time . . . 106
24.4 Oscillators . . . 106
24.5 Clock Users . . . 106
24.6 Clock Events . . . 106
24.7 Controlling Clocks According to a Scenario . . . 106

25 The Physical Layer **109**
25.1 Overview . . . 109
25.2 Generic Radio . . . 109
25.3 Components of a Radio . . . 109
25.4 Layered Radio Models . . . 111
25.5 Notable Radio Models . . . 112

26 The Transmission Medium **115**
26.1 Overview . . . 115
26.2 RadioMedium . . . 115
26.3 Propagation Models . . . 116
26.4 Path Loss Models . . . 116
26.5 Obstacle Loss Models . . . 117
26.6 Background Noise Models . . . 117
26.7 Medium Analog Models . . . 117
26.8 Neighbor Cache . . . 118
26.9 Medium Limit Cache . . . 118
26.10 Communication Cache . . . 118
26.11 Improving Scalability . . . 119

27 The Physical Environment **121**
27.1 Overview . . . 121
27.2 PhysicalEnvironment . . . 121
27.3 Physical Objects . . . 121
27.4 Ground Models . . . 122
27.5 Geographic Coordinate System Models . . . 123
27.6 Object Cache . . . 123

28 Node Mobility **125**
28.1 Overview . . . 125
28.2 Built-In Mobility Models . . . 126

29 Modeling Power Consumption **133**
29.1 Overview . . . 133
29.2 Energy Consumer Models . . . 133
29.3 Energy Generator Models . . . 134
29.4 Energy Storage Models . . . 134
29.5 Energy Management Models . . . 135

30 Network Emulation **137**
30.1 Motivation . . . 137
30.2 Overview . . . 137
30.3 Preparation . . . 138
30.4 Configuring . . . 139

31 Scenario Scripting **141**
31.1 Overview . . . 141

31.2 ScenarioManager . . . 141

32 Modeling Node Failures **143**
32.1 Overview . . . 143
32.2 NodeStatus . . . 144
32.3 Scripting . . . 144

33 Packet Filter Expressions **145**

34 Collecting Results **147**
34.1 Recording Statistics . . . 147
34.2 Measuring along Packet Flows . . . 147
34.3 Recording PCAP Traces . . . 148
34.4 Recording Routing Tables . . . 149

35 Visualization **151**
35.1 Overview . . . 151
35.2 Visualizing Network Communication . . . 151
35.3 Visualizing The Infrastructure . . . 155
35.4 Visualizing Satellite and Geographic Scenarios . . . 155

36 Instrument Figures **157**
36.1 Overview . . . 157
36.2 Instrument Types . . . 157
36.3 Using Instrument Figures . . . 158
36.4 Instrument Figure Attributes . . . 159

37 Appendix: Author's Guide **161**
37.1 Overview . . . 161
37.2 Guidelines . . . 161

38 History **163**
38.1 IPSuite to INET Framework (2000-2006) . . . 163

Release: 4.7

This manual is written for users who are interested in assembling simulations using the components provided by the INET Framework. (In contrast, if you need to modify INET's components or plan to extend INET with new protocols or other components using C++, the INET Developer's Guide is recommended.)

CHAPTER

ONE

INTRODUCTION

1.1 What is INET Framework

The INET Framework is an open-source model library for the OMNeT++ simulation environment. Protocols, agents, and other models are provided for researchers and students working with communication networks. INET is especially useful for designing and validating new protocols or exploring new or exotic scenarios.

A wide class of communication networks are supported by INET, including wired, wireless, mobile, ad hoc, and sensor networks. Models for the Internet stack (TCP, UDP, IPv4, IPv6, OSPF, BGP, etc.), link layer protocols (Ethernet, PPP, IEEE 802.11, various sensor MAC protocols, etc.), refined support for the wireless physical layer, MANET routing protocols, DiffServ, MPLS with LDP and RSVP-TE signaling, several application models, and many other protocols and components are included. Support is also provided for node mobility, advanced visualization, network emulation, and more.

INET serves as a base for several other simulation frameworks which extend it into specific directions, such as vehicular networks, overlay/peer-to-peer networks, or LTE.

1.2 Designed for Experimentation

INET is built around the concept of modules that communicate through message passing. Components represent agents and network protocols, which can be freely combined to form hosts, routers, switches, and other networking devices. Users can program new components, and existing components are designed to be easy to understand and modify.

INET benefits from the infrastructure provided by OMNeT++. In addition to making use of the services provided by the OMNeT++ simulation kernel and library (component model, parameterization, result recording, etc.), models can be developed, assembled, parameterized, run, and their results evaluated from the comfort of the OMNeT++ Simulation IDE or from the command line.

INET Framework is maintained by the OMNeT++ team for the community, utilizing patches and new models contributed by members of the community.

1.3 Scope of this Manual

This manual is written for users who are interested in assembling simulations using the components provided by the INET Framework. (In contrast, if one is interested in modifying INET's components or plans to extend INET with new protocols or other components using C++, the *INET Developers Guide* is recommended.)

This manual does not attempt to be a reference for INET. It focuses on conveying the big picture and does not attempt to cover all components or try to document the parameters, gates, statistics, or precise operation of individual components. For such information, users should refer to the *INET Reference*, a web-based cross-referenced documentation generated from NED and MSG files.

A working knowledge of OMNeT++ is assumed.

CHAPTER

TWO

USING THE INET FRAMEWORK

2.1 Installation

There are several ways to install the INET Framework:

- Allow the OMNeT++ IDE download and install it for you. This is the easiest way. Just accept the offer to install INET in the dialog that comes up when the IDE is started for the first time. Alternatively, it can be installed by choosing *Help → Install Simulation Models* at any later time.
- The latest stable version of the INET Framework compatible with your version of OMNeT++ can be installed from the INET Framework website at *http://inet.omnetpp.org*. If a different version is required, it can also be downloaded from the website. Installation instructions are provided on the website.
- The INET Framework can be cloned from the GitHub repository. If you are familiar with *git*, clone the INET Framework project repository (`inet-framework/inet`) and check out the chosen revision. Then follow the `INSTALL` file in the project root.

2.2 Installing INET Extensions

If INET extensions (e.g. Veins or SimuLTE) are planned to be used, the installation instructions provided with them should be followed.

Usually, the following procedure works if no specific instructions are available:

- First, it should be checked if the project root contains a file named `.project`.
- If the file exists, the project can be imported into the IDE by using *File → Import → General → Existing Project* Make sure that the project is recognized as an OMNeT++ project the *Project Properties* dialog contains a page titled *OMNeT++*), and it lists the INET project as a dependency (check the *Project References* page in the *Project Properties* dialog).
- If there is no `.project` file, create an empty OMNeT++ project using the *New OMNeT++ Project* wizard in *File → New*. Then add the INET project as a dependency using the *Project References* page in the *Project Properties* dialog, and copy the source files into the project.

2.3 Getting Familiar with INET

The INET Framework is based on OMNeT++ and follows the same concept of modules communicating through message passing. OMNeT++ compound modules are used to represent hosts, routers, switches, and other network devices. These compound modules are made up of simple modules that represent protocols, applications, and other functional units. A network consists of an OMNeT++ compound module that contains host, router, and other modules.

Modules are organized into a directory structure that aligns with the OSI layers:

- `src/inet/applications/` – traffic generators and application models
- `src/inet/transportlayer/` – transport layer protocols
- `src/inet/networklayer/` – network layer protocols and accessories

- src/inet/linklayer/ – link layer protocols and accessories
- src/inet/physicallayer/ – physical layer models
- src/inet/routing/ – routing protocols (internet and ad hoc)
- src/inet/mobility/ – mobility models
- src/inet/power/ – energy consumption modeling
- src/inet/environment/ – model of the physical environment
- src/inet/node/ – preassembled network node models
- src/inet/visualizer/ – visualization components (2D and 3D)
- src/inet/common/ – miscellaneous utility components

The hierarchical package names in the OMNeT++ NED language correspond to packages in directories under src/. For example, the src/inet/transportlayer/tcp directory corresponds to the inet.transportlayer.tcp NED package.

For modularity, the INET Framework has about 80 *project features*, which are parts of the codebase that can be disabled as a unit. After installation, not all project features are enabled in the default setup. The list of available project features can be reviewed in the *Project* → *Project Features...* dialog in the IDE. To learn more about project features, refer to the *OMNeT++ User Guide*.

CHAPTER

THREE

NETWORKS

3.1 Overview

INET heavily builds upon the modular architecture of OMNeT++. It provides numerous domain-specific and highly parameterizable components that can be combined in many ways. The primary means of building large custom network simulations in INET is the composition of existing models with custom models, starting from small components and gradually forming ever larger ones up until the composition of the network. Users are not required to have programming experience to create simulations unless they also want to implement their own protocols, for example.

Assembling an INET simulation starts with defining a module that represents the network. Networks are compound modules that contain network nodes, automatic network configurators, and sometimes additionally a transmission medium, physical environment, various visualizers, and other infrastructure-related modules. Networks also contain connections between network nodes representing cables. Large hierarchical networks may be further organized into compound modules to directly express the hierarchy.

There are no predefined networks in INET because it is very easy to create one and because of the vast possibilities. However, the OMNeT++ IDE provides several topology generator wizards for advanced scenarios.

As INET is an OMNeT++-based framework, users mainly use NED to describe the model topology and ini files to provide configuration.

3.2 Built-in Network Nodes and Other Top-Level Modules

INET provides several pre-assembled network nodes with carefully selected components. They support customization via parameters and parametric submodule types, but they are not meant to be universal. Sometimes it may be necessary to create special network node models for particular simulation scenarios. In any case, the following list gives a taste of the built-in network nodes.

- `StandardHost` contains the most common Internet protocols: UDP, TCP, IPv4, IPv6, Ethernet, IEEE 802.11. It also supports an optional mobility model, optional energy models, and any number of applications that are entirely configurable from INI files.
- `EthernetSwitch` models an Ethernet switch containing a relay unit and one MAC unit per port.
- `Router` provides the most common routing protocols: OSPF, BGP, RIP, PIM.
- `AccessPoint` models a Wifi access point with multiple IEEE 802.11 network interfaces and multiple Ethernet ports.
- `WirelessHost` provides a network node with one (default) IEEE 802.11 network interface in infrastructure mode, suitable for use with an `AccessPoint`.
- `AdhocHost` is a `WirelessHost` with the network interface configured in ad-hoc mode and forwarding enabled.

Network nodes communicate at the network level by exchanging OMNeT++ messages, which are the abstract representations of physical signals on the transmission medium. Signals are either sent through OMNeT++ connections

in the wired case or sent directly to the gate of the receiving network node in the wireless case. Signals encapsulate INET-specific packets that represent the transmitted digital data. Packets are further divided into chunks that provide alternative representations for smaller pieces of data (e.g., protocol headers, application data).

Additionally, there are components that occur on the network level, but they are not models of physical network nodes. They are necessary to model other aspects. Some of them are:

- A *radio medium* module such as `Ieee80211RadioMedium`, `ScalarRadioMedium`, and `UnitDiskRadioMedium` (there are a few of them) are a required component of wireless networks.
- `PhysicalEnvironment` models the effect of the physical environment (i.e., obstacles) on radio signal propagation. It is an optional component.
- *Configurators*, such as `Ipv4NetworkConfigurator`, `Ipv6NetworkConfigurator`, `L2NetworkConfigurator`, and `NextHopNetworkConfigurator`, configure various aspects of the network. For example, `Ipv4NetworkConfigurator` (and its IPv6 counterpart `Ipv6NetworkConfigurator`) assigns IP addresses to hosts and routers and sets up static routing. It is used when modeling dynamic IP address assignment (e.g., via DHCP) or when dynamic routing is not of importance. `L2NetworkConfigurator` allows one to configure 802.1 LANs and provide STP/RSTP-related parameters such as link cost, port priority, and the "is-edge" flag. Time-Sensitive Networking adds further configurators for gate scheduling, stream redundancy and failure protection (see *Time-Sensitive Networking*).
- `ScenarioManager` allows scripted scenarios, such as timed failure and recovery of network nodes.
- *Group coordinators* are needed for the operation of some group mobility models. For example, `MoBanCoordinator` is the coordinator module for the MoBAN mobility model.
- *Visualizers*, like `PacketDropOsgVisualizer`, provide graphical rendering of some aspect of the simulation, either in 2D (canvas) or 3D (using OSG or osgEarth). The usual choice is `IntegratedVisualizer`, which bundles together an instance of each specific visualizer type in a compound module.

3.3 Typical Networks

3.3.1 Wired Networks

Wired network connections, for example, Ethernet cables, are represented with standard OMNeT++ connections using the `DatarateChannel` NED type. The channel's `datarate` and `delay` parameters must be provided for all wired connections. The number of wired interfaces in a host (or router) usually does not need to be manually configured because it can be automatically inferred from the actual number of links to neighbor nodes.

The following example shows how straightforward it is to create a model for a simple wired network. This network contains a server connected to a router using PPP, which in turn is connected to a switch using Ethernet. The network also contains a parameterizable number of clients, all connected to the switch, forming a star topology. The utilized network nodes are all predefined modules in INET. To avoid the manual configuration of IP addresses and routing tables, an automatic network configurator is also included.

```
network WiredNetworkExample
{
    parameters:
        int numClients; // number of clients in the network
    submodules:
        configurator: Ipv4NetworkConfigurator; // network autoconfiguration
        server: StandardHost; // predefined standard host
        router: Router; // predefined router
        switch: EthernetSwitch; // predefined ethernet switch
        client[numClients]: StandardHost;
    connections: // network level connections
        router.pppg++ <--> { datarate = 1GBps; } <--> server.pppg++; // PPP
        switch.ethg++ <--> Eth1G <--> router.ethg++; // bidirectional ethernet
        for i=0..numClients-1 {
            client[i].ethg++ <--> Eth1G <--> switch.ethg++; // ethernet
```

(continues on next page)

(continued from previous page)

```
        }
}
```

In order to run a simulation using the above network, an OMNeT++ INI file must be created. The INI file selects the network, sets its number of clients parameter, and configures a simple TCP application for each client. The server is configured to have a TCP application that echos back all data received from the clients individually.

```
network = WiredNetworkExample
*.numClients = 10 # number of clients in network
*.client[*].numApps = 1 # number of applications on clients
*.client[*].app[0].typename = "TcpSessionApp" # client application type
*.client[*].app[0].connectAddress = "server" # destination address
*.client[*].app[0].connectPort = 1000 # destination port
*.client[*].app[0].sendBytes = 1MB # amount of data to send
*.server.numApps = 1 # number of applications on server
*.server.app[0].typename = "TcpEchoApp" # server application type
*.server.app[0].localPort = 1000 # TCP server listen port
```

When the above simulation is run, each client application connects to the server using a TCP socket. Then each one of them sends 1MB of data, which in turn is echoed back by the server, and the simulation concludes. The default statistics are written to the `results` folder of the simulation for later analysis.

3.3.2 Wireless Networks

Wireless network connections are not modeled with OMNeT++ connections due to the dynamically changing nature of connectivity. For wireless networks, an additional module, one that represents the transmission medium, is required to maintain connectivity information.

```
network WirelessNetworkExample
    submodules:
        configurator: Ipv4NetworkConfigurator;
        radioMedium: Ieee80211ScalarRadioMedium;
        host1: WirelessHost { @display("p=200,100"); }
        host2: WirelessHost { @display("p=500,100"); }
        accessPoint: AccessPoint { @display("p=374,200"); }
}
```

In the above network, positions in the display strings provide positions for the transmission medium during the computation of signal propagation and path loss.

In addition, `host1` is configured to periodically send UDP packets to `host2` over the AP.

```
network = WirelessNetworkExample
*.host1.numApps = 1
*.host1.app[0].typename = "UdpBasicApp"
*.host1.app[0].destAddresses = "host2"
*.host1.app[0].destPort = 1000
*.host1.app[0].messageLength = 100Byte
*.host1.app[0].sendInterval = 100ms
*.host2.numApps = 1
*.host2.app[0].typename = "UdpSink"
*.host2.app[0].localPort = 1000
**.arp.typename = "GlobalArp"
**.netmaskRoutes = ""
```

3.3.3 Mobile Ad hoc Networks

```
network MobileAdhocNetworkExample
{
    parameters:
        int numHosts; // number of nodes in the network
    submodules:
        configurator: Ipv4NetworkConfigurator; // network autoconfiguration
        radioMedium: Ieee80211ScalarRadioMedium; // 802.11 physical medium
        host[numHosts]: AdhocHost; // ad-hoc wifi nodes
}
```

```
network = MobileAdhocNetworkExample
*.numHosts = 10 # number of hosts in the MANET
*.host[*].mobility.typename = "MassMobility" # stochastic mobility model
*.host[*].mobility.initFromDisplayString = false # ignore display string
*.host[*].mobility.changeInterval = truncnormal(2s, 0.5s) # between turns
*.host[*].mobility.angleDelta = normal(0deg, 30deg) # random turn
*.host[*].mobility.speed = truncnormal(20mps, 8mps) # random speed
*.host[*].numApps = 1 # number of applications on hosts
*.host[*].app[0].typename = "PingApp" # application type for all hosts
*.host[*].app[0].destAddr = "host[0]" # ping destination
*.host[*].app[0].startTime = uniform(1s,5s) # to avoid synchronization
*.host[*].app[0].printPing = true # print usual ping results to stdout
```

3.4 Frequent Tasks (How To...)

This section contains quick and somewhat superficial advice on many practical tasks.

3.4.1 Automatic Wired Interfaces

In many wired network simulations, the number of wired interfaces does not need to be manually configured because it can be automatically inferred from the actual number of connections between network nodes.

```
router1.ethg++ <--> Eth1G <--> router2.ethg++; // automatic interfaces
```

3.4.2 Multiple Wireless Interfaces

All built-in wireless network nodes support multiple wireless interfaces, but only one is enabled by default.

```
*.host[*].numWlanInterfaces = 2 # number of wireless network interfaces
*.host[*].wlan[0].agent.defaultSsid = "alpha" # connects to alpha network
*.host[*].wlan[1].agent.defaultSsid = "bravo" # connects to bravo network
```

3.4.3 Specifying Addresses

Nearly all application layer modules, but several other components as well, have parameters that specify network addresses. They typically accept addresses given with any of the following syntax variations:

- literal IPv4 address: `"186.54.66.2"`
- literal IPv6 address: `"3011:7cd6:750b:5fd6:aba3:c231:e9f9:6a43"`
- module name: `"server"`, `"subnet.server[3]"`
- interface of a host or router: `"server/eth0"`, `"subnet.server[3]/eth0"`
- IPv4 or IPv6 address of a host or router: `"server(ipv4)"`, `"subnet.server[3](ipv6)"`

- IPv4 or IPv6 address of an interface of a host or router: `"server/eth0(ipv4)"`, `"subnet.server[3]/eth0(ipv6)"`

3.4.4 Node Failure and Recovery

3.4.5 Enabling Dual IP Stack

All built-in network nodes support dual Internet protocol stacks, that is, both IPv4 and IPv6 are available. They are also supported by transport layer protocols, link layer protocols, and most applications. Only IPv4 is enabled by default, so in order to use IPv6, it must be enabled first, and an application supporting IPv6 (e.g., `PingApp`) must be used. The following example shows how to configure two ping applications in a single node where one is using an IPv4 and the other is using an IPv6 destination address.

```
*.host[*].hasIpv4 = true # enable IPv4 protocol
*.host[*].hasIpv6 = true # enable IPv6 protocol
*.host[*].numApps = 2 # number of applications on hosts
*.host[*].app[*].typename = "PingApp" # type for both applications
*.host[*].app[0].destAddr = "host[0](ipv4)" # uses IPv4 detination address
*.host[*].app[1].destAddr = "host[0](ipv6)" # uses IPv6 detination address
```

3.4.6 Enabling Packet Forwarding

In general, network nodes don't forward packets by default, only `Router` and the like do. Nevertheless, it's possible to enable packet forwarding as simply as flipping a switch.

```
*.host[*].forwarding = true
```

CHAPTER

FOUR

NETWORK NODES

4.1 Overview

Hosts, routers, switches, access points, mobile phones, and other network nodes are represented in INET with compound modules. A few node types like `StandardHost`, `Router` were introduced in the previous chapter, and networks were demonstrated to be built from them. In this chapter, the internals of these node models will be explored to provide a deeper understanding of their customization possibilities and to give some guidance on assembling custom node models.

4.2 Ingredients

Node models are assembled from other modules representing applications, communication protocols, network interfaces, routing tables, mobility models, energy models, and other functionalities. These modules can be categorized into the following:

- *Applications* often models user behavior, the application program (e.g., browser), and the application layer protocol (e.g., HTTP). Applications typically utilize transport layer protocols (e.g., TCP and/or UDP), but they may also directly use lower layer protocols (e.g., IP or Ethernet) via sockets.
- *Routing protocols* are provided as separate modules: OSPF, BGP, and AODV for MANET routing. These modules use TCP, UDP, and IPv4, and manipulate routes in the `Ipv4RoutingTable` module.
- *Transport layer protocols* are connected to applications and network layer protocols. They are most often represented by simple modules, currently TCP, UDP, and SCTP are supported. TCP has several implementations: `Tcp` is the OMNeT++ native implementation; `TcpLwip` module wraps the lwIP TCP stack; and `TcpNsc` module wraps the Network Simulation Cradle library.
- *Network layer protocols* are connected to transport layer protocols and network interfaces, usually modeled as compound modules: `Ipv4NetworkLayer` for IPv4, and `Ipv6NetworkLayer` for IPv6. The `Ipv4NetworkLayer` module contains several protocol modules: `Ipv4`, `Arp`, `Icmp`, and `Icmpv6`.
- *Network interfaces* are represented by compound modules which are connected to the network layer protocols and other network interfaces in the wired case. They are usually modeled as compound modules containing separate modules for queues, classifiers, MAC, and PHY protocols.
- *Link layer protocols* are usually simple modules sitting in network interface modules. Some protocols, for example IEEE 802.11 MAC, are modeled as compound modules themselves due to the complexity of the protocol.
- *Physical layer protocols* are compound modules also being part of network interface modules.
- *Interface table* maintains the set of network interfaces (e.g. `eth0`, `wlan0`) in the network node. Interfaces are registered dynamically during the initialization of network interfaces.
- *Routing tables* maintain the list of routes for the corresponding network protocol (e.g., `Ipv4RoutingTable` for `Ipv4`). Routes are added by automatic network configurators or routing protocols. Network protocols use the routing tables to determine the best matching route for datagrams.
- *Mobility modules* are responsible for moving around the network node in the simulated scene. The mobility model is mandatory for wireless simulations even if the network node is stationary. The mobility module

stores the location of the network node which is needed to compute wireless propagation and path loss. Different mobility models are provided as different modules. Network nodes define their mobility submodule with a parametric type, so the mobility model can be changed in the configuration.

- *Energy modules* model energy storage mechanisms, energy consumption of devices and software processes, energy generation of devices, and energy management processes that shutdown and startup network nodes.
- *Status* (`NodeStatus`) keeps track of the status of the network node (up, down, etc.)
- *Other modules* with particular functionality such as `PcapRecorder`, are also available.

4.3 Node Architecture

Within network nodes, OMNeT++ connections are used to represent communication opportunities between protocols. Packets and messages sent on these connections represent software or hardware activity.

Although protocols may also be connected to each other directly, in most cases, they are connected via *dispatcher modules*. Dispatchers (`MessageDispatcher`) are small, low-overhead modules that allow protocol components to be connected in one-to-many and many-to-many fashions, ensuring that messages and packets sent from one component end up being delivered to the correct component. Dispatchers require no manual configuration, as they use discovery and peek into packets.

In their pre-assembled node models, dispatchers allow arbitrary protocol components to communicate directly with each other, not just ones in neighboring layers.

4.4 Customizing Nodes

The built-in network nodes are designed to be as versatile and customizable as possible. This is achieved through several ways:

4.4.1 Submodule and Gate Vectors

One way is the usage of gate vectors and submodule vectors. The sizes of vectors may come from parameters or be derived from the number of external connections to the network node. For example, a host may have an arbitrary number of wireless interfaces, and it will automatically have as many Ethernet interfaces as the number of Ethernet devices connected to it.

Wireless interfaces for hosts are defined as follows:

```
wlan[numWlanInterfaces]: <snip> // wlan interfaces in StandardHost etc al.
```

Where `numWlanInterfaces` is a module parameter with a default value of either 0 or 1 (this is different for e.g. `StandardHost` and ned:*WirelessHost*.) To configure a host to have two interfaces, add the following line to the ini file:

```
**.hostA.numWlanInterfaces = 2
```

4.4.2 Conditional Submodules

Submodules that are not vectors are often conditional. For example, the TCP protocol module in hosts is conditional on the `hasTcp` parameter. Thus, to disable TCP support in a host (it is enabled by default), use the following ini file line:

```
**.hostA.hasTcp = false
```

4.4.3 Parametric Types

Another often-used way of customization is parametric types, where the type of a submodule (or a channel) may be specified as a string parameter. Almost all submodules in the built-in node types have parametric types. For example, the TCP protocol module is defined as follows:

```
tcp: <default("Tcp")> like ITcp if hasTcp;
```

The `typename` parameter of the tcp submodule defaults to the default implementation, `Tcp`. To use another implementation instead, add the following line to the ini file:

```
**.host*.tcp.typename = "TcpLwip"  # use lwIP's TCP implementation
```

Submodule vectors with parametric types are defined without the use of a module parameter to allow elements to have different types. An example is how applications are defined in hosts:

```
app[numApps]: <> like IApp;  // applications in StandardHost et al.
```

And applications can be added in the following way:

```
**.hostA.numApps = 2
**.hostA.apps[0].typename = "UdpBasicApp"
**.hostA.apps[1].typename = "PingApp"
```

4.4.4 Inheritance

Inheritance can be used to derive new, specialized node types from existing ones. A derived NED type may add new parameters, gates, submodules, or connections and may set inherited unassigned parameters to specific values.

For example, `WirelessHost` is derived from `StandardHost`, as shown below:

```
module WirelessHost extends StandardHost
{
    @display("i=device/wifilaptop");
    numWlanInterfaces = default(1);
}
```

4.5 Custom Network Nodes

Despite the many pre-assembled network nodes and the several available customization options, sometimes it is just easier to build a network node from scratch. The following example demonstrates how easy it is to build a simple network node.

This network node already contains a configurable application and several standard protocols. It also demonstrates the usage of the packet dispatching mechanism that is required to connect multiple protocols in a many-to-many relationship.

```
module NetworkNodeExample
{
    parameters:
    gates:
        inout ethg; // ethernet interface connector
        input radioIn; // incoming radio frames from physical medium
    submodules:
        app: <> like IApp; // configurable application
        tcp: Tcp; // standard TCP protocol
        ip: Ipv4; // standard IP protocol
        md: MessageDispatcher; // connects multiple interfaces to IP
```

(continues on next page)

(continued from previous page)

```
        wlan: Ieee80211Interface; // standard wifi interface
        eth: EthernetInterface; // standard ethernet interface
        interfaceTable: InterfaceTable;
    connections: // network node internal connections
        app.socketOut --> tcp.appIn; // application sends data stream
        app.socketIn <-- tcp.appOut; // application receives data stream
        tcp.ipOut --> ip.transportIn; // TCP sends segments
        tcp.ipIn <-- ip.transportOut; // TCP receives segments
        ip.queueOut --> md.in++; // IP sends datagrams
        ip.queueIn <-- md.out++; // IP receives datagrams
        md.out++ --> wlan.upperLayerIn;
        md.in++ <-- wlan.upperLayerOut;
        md.out++ --> eth.upperLayerIn;
        md.in++ <-- eth.upperLayerOut;
        eth.phys <--> ethg; // Ethernet sends frames to cable
        radioIn --> wlan.radioIn; // IEEE 802.11 sends frames to medium
}
```

CHAPTER

FIVE

NETWORK INTERFACES

5.1 Overview

In INET simulations, network interface modules are the primary means of communication between network nodes. They represent the required combination of software and hardware elements from an operating system point-of-view.

Network interfaces are implemented with OMNeT++ compound modules that conform to the `INetworkInterface` module interface. Network interfaces can be further categorized as wired and wireless; they conform to the `IWiredInterface` and `IWirelessInterface` NED types, respectively, which are subtypes of `INetworkInterface`.

5.2 Built-in Network Interfaces

INET provides pre-assembled network interfaces for several standard protocols, protocol tunneling, hardware emulation, etc. The most commonly used network interfaces are as follows.

- `EthernetInterface` represents an Ethernet interface
- `PppInterface` is for wired links using PPP
- `Ieee80211Interface` represents a Wi-Fi (IEEE 802.11) interface
- `Ieee802154NarrowbandInterface` and `Ieee802154UwbIrInterface` represent an IEEE 802.15.4 interface
- `BMacInterface`, `LMacInterface`, `XMacInterface` provide low-power wireless sensor MAC protocols along with a simple hypothetical PHY protocol
- `TunInterface` is a tunneling interface that can be directly used by applications
- `LoopbackInterface` provides local loopback within the network node
- `ExtLowerEthernetInterface` represents a real-world interface, suitable for hardware-in-the-loop simulations

5.3 Anatomy of Network Interfaces

Within the INET Framework, network interfaces are compound modules containing many more components than just the corresponding layer 2 protocol implementation. By default, most of these components are absent but can be added via configuration.

Typical ingredients include:

- *Layer 2 protocol implementation*. For some interfaces such as `PppInterface`, this is a single module. For others like Ethernet and Wi-Fi, separate modules for MAC, LLC, and possibly other subcomponents exist.
- *PHY model*. Some interfaces also contain separate module(s) that implement the physical layer. For example, `Ieee80211Interface` contains a radio module.

- *Output queue*. This module allows experimentation with different queueing policies and implementation of QoS, RED, etc.
- *Traffic conditioners* allow the addition of traffic shaping and policing elements to the interface. Examples include implementing a Diffserv router.
- *Hooks* allow the insertion of extra modules in the incoming and outgoing paths of packets.

5.3.1 Internal vs External Output Queue

Network interfaces usually have a queue module defined with a parametric type like this:

```
queue: <default("DropTailQueue")> like IPacketQueue;
```

When the `typename` parameter of the queue submodule is unspecified (this is the default), the queue module is a `DropTailQueue`. Conceptually, the queue is of infinite size, but for better diagnostics, a hard limit for queue length can often be specified via module parameters. If this limit is exceeded, the simulation stops with an error.

If the `typename` parameter of the queue module is not empty, it must name an NED type implementing the `IPacketQueue` interface. The queue module model enables the modeling of finite buffers or the implementation of various queueing policies for QoS and/or RED.

The most frequently used module type for the queue module is `DropTailQueue`, a finite-size FIFO that drops overflowing packets). Other queue types implementing queueing policies can be created by assembling compound modules from queueing model and DiffServ components (see chapter *Differentiated Services*). One such example of a compound module is `DiffservQueue`.

Below is an example ini file fragment that installs a priority queue on PPP interfaces:

```
**.ppp[*].ppp.queue.typename = "PriorityQueue"
**.ppp[*].ppp.queue.packetCapacity = 10
**.ppp[*].ppp.queue.numQueues = 2
**.ppp[*].ppp.queue.classifier.typename = "WrrClassifier"
**.ppp[*].ppp.queue.classifier.weights = "1 1"
```

5.3.2 Traffic Conditioners

Many network interfaces contain optional traffic conditioner submodules defined with parametric types, like this:

```
ingressTC: <default("")> like ITrafficConditioner if typename != "";
egressTC: <default("")> like ITrafficConditioner if typename != "";
```

Traffic conditioners allow the implementation of the policing and shaping actions of a Diffserv router. They can be added to the input or output packets paths in the network interface. (On the output path, they are added before the queue module.)

Traffic conditioners must implement the `ITrafficConditioner` module interface. Traffic conditioners can be assembled from DiffServ components (see chapter *Differentiated Services*). INET does not provide any preassembled traffic conditioners, but some can be found in the example simulations.

Example configuration with fictitious types:

```
**.ppp[*].ingressTC.typename = "CustomIngressTC"
**.ppp[*].egressTC.typename = "CustomEgressTC"
```

5.4 The Interface Table

Network nodes typically contain an `InterfaceTable` module. The interface table serves as a registry of all network interfaces in the host. It does not send or receive messages; other modules access it via C++ function calls. The interface table's contents can also be inspected, for example, in Qtenv.

Network interfaces register themselves in the interface table at the beginning of the simulation, typically through the MAC (or equivalent) module.

5.5 Wired Network Interfaces

Wired interfaces have special-purpose OMNeT++ gates representing the capability of having an external physical connection to another network node (e.g., an Ethernet port). In order to enable wired communication, these gates must be connected with special connections representing the physical cable between the physical ports. The connections must use special OMNeT++ channels (e.g., `DatarateChannel`) which determine data rate and delay parameters.

Wired network interfaces are compound modules implementing the `IWiredInterface` interface. The following wired network interfaces are available in INET.

5.5.1 PPP

Chapter *Point-to-Point Links* describes network interfaces for point-to-point links (namely `PppInterface`). They are typically used in routers.

5.5.2 Ethernet

Ethernet interfaces (`EthernetInterface`), together with models of Ethernet devices such as switches and hubs, are described in chapter *The Ethernet Model*.

5.6 Wireless Network Interfaces

Wireless interfaces use direct sending[1] for communication instead of links. As a result, their compound modules do not have output gates at the physical layer; only an input gate dedicated to receiving is present. Another difference from the wired case is that wireless interfaces require (and collaborate with) a *transmission medium* module at the network level. The medium module represents the shared transmission medium (electromagnetic field or an acoustic medium) and is responsible for modeling physical effects such as signal attenuation and maintaining connectivity information. While wired interfaces can function without explicit modeling of the physical layer, a PHY module is an indispensable part of a wireless interface.

Wireless network interfaces are compound modules that implement the `IWirelessInterface` interface. The following sections provide an overview of the wireless interfaces available in INET.

5.6.1 Generic Wireless Interface

The `WirelessInterface` compound module is a generic implementation of `IWirelessInterface`. The types of the MAC protocol and PHY layer (the radio) are parameters in this network interface:

```
mac: <> like IMacProtocol;
radio: <> like IRadio if typename != "";
```

There are specialized versions of `WirelessInterface` where the MAC and the radio modules are fixed to a particular value. One example is `BMacInterface`, which contains a `BMac` and an `ApskRadio`.

5.6.2 IEEE 802.11

Chapter *The 802.11 Model* covers IEEE 802.11 or Wi-Fi network interfaces (`Ieee80211Interface`), as well as models of devices acting as access points (AP).

[1] OMNeT++ `sendDirect()` calls

5.6.3 IEEE 802.15.4

A separate chapter, *The 802.15.4 Model*, covers `Ieee802154NarrowbandInterface`.

5.6.4 Wireless Sensor Networks

Chapter *MAC Protocols for Wireless Sensor Networks* covers MAC protocols for wireless sensor networks (WSNs) and the corresponding network interfaces.

5.6.5 CSMA/CA

`CsmaCaMac` implements an imaginary CSMA/CA-based MAC protocol with optional acknowledgements and a retry mechanism. With the appropriate settings, it can approximate basic 802.11b ad-hoc mode operation.

`CsmaCaMac` provides a lot of room for experimentation: acknowledgements can be turned on/off, and operation parameters like inter-frame gap sizes, backoff behavior (slot time, minimum and maximum number of slots), maximum retry count, header and ACK frame sizes, bit rate, etc. can be configured via NED parameters.

`CsmaCaInterface` interface is a `WirelessInterface` with the MAC type set to `CsmaCaMac`.

5.6.6 Acking MAC

Not every simulation requires a detailed simulation of the lower layers. `AckingWirelessInterface` is a highly abstracted wireless interface that offers simplicity in scenarios where Layer 1 and 2 effects can be completely ignored (e.g., testing the basic functionality of a wireless ad-hoc routing protocol).

`AckingWirelessInterface` is a `WirelessInterface` parameterized to contain a unit disk radio (`GenericUnitDiskRadio`) and a trivial MAC protocol (`AckingMac`).

The most important parameter `GenericUnitDiskRadio` accepts is the transmission range. When a radio transmits a frame, all other radios within transmission range are able to receive the frame correctly, and radios that are out of range will not be affected at all. Interference modeling (collisions) is optional, and it is recommended to turn it off with `AckingMac`.

`AckingMac` implements a trivial MAC protocol that has packet encapsulation and decapsulation, but no real medium access procedure. Frames are simply transmitted on the wireless channel as soon as the transmitter becomes idle. There is no carrier sense, collision avoidance, or collision detection. `AckingMac` also provides an optional out-of-band acknowledgement mechanism (using C++ function calls, not actual wirelessly sent frames), which is turned on by default. There is no retransmission: if the acknowledgement does not arrive after the first transmission, the MAC gives up and counts the packet as a failed transmission.

5.6.7 Shortcut

`ShortcutMac` supports error-free "teleportation" of packets to the peer MAC entity, with delay computed from transmission duration and propagation delay. The physical layer is completely bypassed in this model. The corresponding network interface type, `ShortcutInterface`, does not have a radio model.

`ShortcutInterface` is useful for modeling wireless networks where full connectivity is assumed, and Layer 1 and Layer 2 effects can be completely ignored.

5.7 Special-Purpose Network Interfaces

5.7.1 Tunnelling

`TunInterface` is a virtual network interface that can be used for creating tunnels. However, it is more powerful than that. It allows an application-layer module to capture packets sent to the TUN interface and perform any desired action on them, including sending them to a peer entity in a UDP or plain IPv4 packet.

To set up a tunnel, add an instance of `TunnelApp` to the node, and specify the protocol (IPv4 or UDP) and the remote endpoint of the tunnel (address and port) in the parameters.

5.7.2 Local Loopback

`LoopbackInterface` provides local loopback within the network node.

5.7.3 External Interfaces

`ExtLowerEthernetInterface` represents a real-world interface suitable for hardware-in-the-loop simulations. External interfaces are explained in Chapter *Network Emulation*.

5.8 Custom Network Interfaces

It is also possible to build custom network interfaces. The following example shows how to build a custom wireless interface.

```
module WirelessInterfaceExample
{
    gates:
        input upperLayerIn; // packets from network layer in the same host
        output upperLayerOut; // packets to network layer in the same host
        input radioIn; // incoming packets from other hosts in the network
    submodules:
        mac: AckingMac; // simple MAC supporting ACKs
        radio: ApskScalarRadio; // simple radio supporting many modulations
    connections: // network interface internal connections
        mac.upperLayerOut --> upperLayerOut;
        mac.upperLayerIn <-- upperLayerIn;
        radio.upperLayerOut --> mac.lowerLayerIn;
        radio.upperLayerIn <-- mac.lowerLayerOut;
        radioIn --> radio.radioIn;
}
```

The above network interface contains very simple hypothetical MAC and PHY protocols. The MAC protocol provides only acknowledgment without other services (e.g., carrier sense, collision avoidance, collision detection), and the PHY protocol uses one of the predefined APSK modulations for the entire signal (preamble, header, and data) without other services (e.g., scrambling, interleaving, forward error correction).

CHAPTER

SIX

APPLICATIONS

6.1 Overview

This chapter describes application models and traffic generators. All applications implement the `IApp` module interface to ease configuration. For example, `StandardHost` contains an application submodule vector that can be filled in with specific applications from the INI file.

INET applications fall into two categories. In the first category, applications implement very specific behaviors, and generate corresponding traffic patterns based on their specific parameters. These applications are implemented as simple modules.

In the second category, applications are more generic. Traffic generation is separated from the usage of the protocol, such as `Udp` or `Tcp`. These applications are implemented as compound modules. They contain separate configurable traffic source, traffic sink, and protocol input/output submodules. This approach allows building complex traffic patterns by composing queuing model elements.

6.2 TCP applications

This section describes the applications using the TCP protocol. These applications use `GenericAppMsg` objects to represent the data sent between the client and server. The client message contains the expected reply length, the processing delay, and a flag indicating that the connection should be closed after sending the reply. This way intelligence (behavior specific to the modeled application, e.g. HTTP, SMB, database protocol) needs only to be present in the client, and the server model can be kept simple and dumb.

6.2.1 TcpBasicClientApp

A client for a generic request-response style protocol over TCP. It may be used as a rough model of HTTP or FTP users.

The model communicates with the server in sessions. During a session, the client opens a single TCP connection to the server, sends several requests (always waiting for the complete reply to arrive before sending a new request), and closes the connection.

The server app should be `TcpGenericServerApp`; the model sends a `GenericAppMsg` messages.

Example settings:

FTP:

```
numRequestsPerSession = exponential(3)
requestLength = truncnormal(20,5)
replyLength = exponential(1000000)
```

HTTP:

```
numRequestsPerSession = 1 # HTTP 1.0
numRequestsPerSession = exponential(5)  # HTTP 1.1, with keepalive
```

(continues on next page)

(continued from previous page)

```
requestLength = truncnormal(350,20)
replyLength = exponential(2000)
```

Note that since most web pages contain images and may contain frames, applets etc, possibly from various servers, and browsers usually download these items in parallel to the main HTML document, this module cannot serve as a realistic web client.

Also, with HTTP 1.0 it is the server that closes the connection after sending the response, while in this model, it is the client.

6.2.2 TcpSinkApp

This module accepts any number of incoming TCP connections and discards whatever arrives on them.

6.2.3 TcpGenericServerApp

A generic server application for modeling TCP-based request-reply style protocols or applications.

The module accepts any number of incoming TCP connections and expects to receive messages of class `GenericAppMsg` on them. A message should contain how large the reply should be (number of bytes). `TcpGenericServerApp` will just change the length of the received message accordingly and send back the same message object. The reply can be delayed by a constant time (`replyDelay` parameter).

6.2.4 TcpEchoApp

The `TcpEchoApp` application accepts any number of incoming TCP connections and sends back the data that arrives on them. The byte counts are multiplied by `echoFactor` before echoing. The reply can also be delayed by a constant time (`echoDelay` parameter).

6.2.5 TcpSessionApp

A single-connection TCP application: it opens a connection, sends the given number of bytes, and closes. Sending may be one-off, or may be controlled by a "script" which is a series of (time, number of bytes) pairs. It may act either as a client or as a server. Compatible with both IPv4 and IPv6.

Opening the connection

Depending on the type of opening the connection (active/passive), the application may be either a client or a server. In passive mode, the application will listen on the given local local port and wait for an incoming connection. In active mode, the application will bind to the given local local address and local port and connect to the given address and port. It is possible to use an ephemeral port as a local port.

Even when in server mode (passive open), the application will only serve one incoming connection. Further connect attempts will be refused by TCP (it will send RST) for lack of LISTENing connections.

The time of opening the connection is in the `tOpen` parameter.

Sending data

Regardless of the type of OPEN, the application can be made to send data. One way of specifying sending is via the `tSend`, `sendBytes` parameters, the other way is `sendScript`. With the former, `sendBytes` bytes will be sent at `tSend`. When using `sendScript`, the format of the script is:

```
<time> <numBytes>; <time> <numBytes>;...
```

Closing the connection

The application will issue a TCP CLOSE at time `tClose`. If `tClose=-1`, no CLOSE will be issued.

6.2.6 TelnetApp

Models Telnet sessions with a specific user behavior. The server app should be `TcpGenericServerApp`.

In this model, the client repeats the following activity between `startTime` and `stopTime`:

1. Opens a telnet connection
2. Sends `numCommands` commands. The command is `commandLength` bytes long. The command is transmitted as entered by the user character by character, there is `keyPressDelay` time between the characters. The server echoes each character. When the last character of the command is sent (new line), the server responds with a `commandOutputLength` bytes long message. The user waits `thinkTime` interval between the commands.
3. Closes the connection and waits `idleInterval` seconds
4. If the connection is broken, it is noticed after `reconnectInterval`, and the connection is reopened

All parameters in the above description are marked as `volatile`, allowing for using distributions to emulate random behavior.

Note

This module emulates a very specific user behavior, and as such, it should be viewed as an example rather than a generic Telnet model. If you want to model realistic Telnet traffic, you are encouraged to gather statistics from packet traces on a real network and write your model accordingly.

6.2.7 TcpServerHostApp

This module hosts TCP-based server applications. It dynamically creates and launches a new "thread" object for each incoming connection.

Server threads can be implemented in C++. An example server thread class is `TcpGenericServerThread`.

6.2.8 Applications composing TCP traffic

The following TCP modules are provided to allow composing applications with more complex traffic without implementing new C++ modules:

- `TcpClientApp`: generic TCP client application with composable traffic source and traffic sink
- `TcpServerApp`: generic TCP server application with a TCP server listener to create TCP server connections
- `TcpServerConnection`: generic TCP server connection with composable traffic source and traffic sink
- `TcpServerListener`: generic TCP server listener for accepting/rejecting TCP connections and for creating TCP server connections
- `TcpRequestResponseApp`: generic request-response based TCP server application with configurable pre-composed traffic source and traffic sink

There are some applications that model the traffic of the telnet protocol:

- `TelnetClientApp`: telnet client application with configurable pre-composed telnet traffic source and traffic sink
- `TelnetServerApp`: telnet server application with pre-configured TCP server listener to create telnet server connections
- `TelnetServerConnection`: telnet server connection with configurable pre-composed telnet traffic source and traffic sink

6.3 UDP applications

The following UDP-based applications are implemented in INET:

- `UdpBasicApp` sends UDP packets to a given IP address at a given interval
- `UdpBasicBurst` sends UDP packets to the given IP address(es) in bursts or acts as a packet sink
- `UdpEchoApp` is similar to `UdpBasicApp`, but it sends back the packet after reception
- `UdpSink` consumes and prints packets received from the `Udp` module
- `UdpVideoStreamClient`,:ned:*UdpVideoStreamServer* simulates video streaming over UDP

The next sections describe these applications in detail.

6.3.1 UdpBasicApp

The `UdpBasicApp` sends UDP packets to a the IP addresses given in the `destAddresses` parameter. The application sends a message to one of the targets in each `sendInterval` interval. The interval between messages and the message length can be given as a random variable. Before the packet is sent, it is emitted in the signal.

The application simply prints the received UDP datagrams. The signal can be used to detect the received packets.

6.3.2 UdpSink

This module binds a UDP socket to a given local port and prints the source and destination and the length of each received packet.

6.3.3 UdpEchoApp

Similar to `UdpBasicApp`, but it sends back the packet after reception. It accepts only packets with `UdpHeader`, i.e., packets that are generated by another `UdpEchoApp`.

When an echo response is received, it emits a signal.

6.3.4 UdpVideoStreamClient

This module is a video streaming client. It sends one "video streaming request" to the server at time `startTime` and receives a stream from `UdpVideoStreamServer`.

The received packets are emitted by the signal.

6.3.5 UdpVideoStreamServer

This is the video stream server to be used with `UdpVideoStreamClient`.

The server will wait for incoming "video streaming requests". When a request arrives, it draws a random video stream size using the `videoSize` parameter and starts streaming to the client. During streaming, it will send UDP packets of size `packetLen` at every `sendInterval`, until `videoSize` is reached. The parameters `packetLen` and `sendInterval` can be set to constant values to create CBR traffic or to random values (e.g., `sendInterval=uniform(1e-6, 1.01e-6)`) to accommodate jitter.

The server can serve several clients and several streams per client.

6.3.6 UdpBasicBurst

Sends UDP packets to the given IP address(es) in bursts or acts as a packet sink. Compatible with both IPv4 and IPv6.

Addressing

The `destAddresses` parameter can contain zero, one, or more destination addresses, separated by spaces. If there is no destination address given, the module will act as a packet sink. If there are more than one addresses, one of them is randomly chosen, either for the whole simulation run or for each burst or for each packet, depending on the value of the `chooseDestAddrMode` parameter. The `destAddrRNG` parameter controls which (local) RNG is used for randomized address selection. The own addresses will be ignored.

An address may be given in the dotted decimal notation or with the module name. (The `L3AddressResolver` class is used to resolve the address.) You can use the "Broadcast" string as an address for sending broadcast messages.

INET also defines several NED functions that can be useful:

- `moduleListByPath("pattern",...)`:
  Returns a space-separated list of the module names. All modules whose full path matches one of the pattern parameters will be included. The patterns may contain wildcards in the same syntax as in ini files. Example:
- `moduleListByNedType("fully.qualified.ned.type",...)`:
  Returns a space-separated list of the module names with the given NED type(s). All modules whose NED type name occurs in the parameter list will be included. The NED type name is fully qualified. Example:

Examples:

```
**.app[0].destAddresses = moduleListByPath("**.host[*]", "**.fixhost[*]")
**.app[1].destAddresses = moduleListByNedType("inet.nodes.inet.StandardHost")
```

The peer can be *UdpSink* or another *UdpBasicBurst*.

Bursts

The first burst starts at `startTime`. Bursts start by immediately sending a packet; subsequent packets are sent at `sendInterval` intervals. The `sendInterval` parameter can be a random value, e.g., `exponential(10ms)`. A constant interval with jitter can be specified as `1s+uniform(-0.01s,0.01s)` or `uniform(0.99s,1.01s)`. The length of the burst is controlled by the `burstDuration` parameter. (Note that if `sendInterval` is greater than `burstDuration`, the burst will consist of one packet only.) The time between bursts is the `sleepDuration` parameter; this can be zero (zero is not allowed for `sendInterval`). The zero `burstDuration` is interpreted as infinity.

Operation as a sink

When the `destAddresses` parameter is empty, the module receives packets and makes statistics only.

6.3.7 Applications composing UDP traffic

The following UDP modules are provided to allow composing applications with more complex traffic without implementing new C++ modules:

- `UdpApp`: generic UDP application with composable traffic source and traffic sink
- `UdpSourceApp`: generic UDP application with a traffic source
- `UdpSinkApp`: generic UDP application with a traffic sink
- `UdpRequestResponseApp`: generic request-response based UDP server application with configurable pre-composed traffic source and traffic sink

6.4 IPv4/IPv6 traffic generators

The applications described in this section use the services of the network layer only; they do not need transport layer protocols. They can be used with both IPv4 and IPv6.

`IIpvxTrafficGenerator` (prototype) sends IP or IPv6 datagrams to the given address at the specified `sendInterval`. The `sendInterval` parameter can be a constant or a random value (e.g., `exponential(1)`). If

the destAddresses parameter contains more than one address, one of them is randomly chosen for each packet. An address may be given in the dotted decimal notation (or, for IPv6, in the usual notation with colons) or with the module name. (The L3AddressResolver class is used to resolve the address.) To disable the model, set destAddresses to "".

The IpvxTrafGen sends messages with length packetLength. The sent packet is emitted in the signal. The length of the sent packets can be recorded as scalars and vectors.

The IpvxTrafSink can be used as a receiver of the packets generated by the traffic generator. This module emits the packet in the signal and drops it. The rcvdPkBytes and endToEndDelay statistics are generated from this signal.

The IpvxTrafGen can also be the peer of the traffic generators; it handles the received packets exactly like the IpvxTrafSink.

6.5 The PingApp application

The PingApp application generates ping requests and calculates the packet loss and round-trip parameters of the replies.

Start/stop time, sendInterval etc., can be specified via parameters. An address may be given in the dotted decimal notation (or, for IPv6, in the usual notation with colons) or with the module name. (The L3AddressResolver class is used to resolve the address.) To disable sending, specify an empty destAddr.

Every ping request is sent out with a sequence number, and replies are expected to arrive in the same order. Whenever there is a jump in the in the received ping responses' sequence number (e.g., 1, 2, 3, 5), then the missing pings (number 4 in this example) are counted as lost. Then if it still arrives later (that is, a reply with a sequence number smaller than the largest one received so far), it will be counted as an out-of-sequence arrival, and at the same time, the number of losses is decremented. (It is assumed that the packet arrived was counted earlier as a loss, which is true if there are no duplicate packets.)

6.6 Ethernet applications

The inet.applications.ethernet package contains modules for a simple client-server application. The EtherAppClient is a simple traffic generator that periodically sends EtherAppReq messages whose length can be configured. destAddress, startTime, waitType, reqLength, respLength.

The server component of the model (EtherAppServer) responds with an EtherAppResp message of the requested length. If the response does not fit into one Ethernet frame, the client receives the data in multiple chunks.

Both applications have a registerSAP boolean parameter. This parameter should be set to true if the application is connected to the Ieee8022Llc module, which requires the registration of the SAP before sending frames.

Both applications collect the following statistics: sentPkBytes, rcvdPkBytes, endToEndDelay.

The client and server application work with any model that accepts Ieee802Ctrl control info on the packets (e.g., the 802.11 model). The applications should be connected directly to the Ieee8022Llc or an EthernetInterface NIC module.

The model also contains a host component that groups the applications and the LLC and MAC components together (EthernetHost). This node does not contain higher layer protocols; it generates Ethernet traffic directly. By default, it is configured to use half-duplex MAC (CSMA/CD).

CHAPTER

SEVEN

TRANSPORT PROTOCOLS

7.1 Overview

In the OSI reference model, the protocols of the transport layer provide host-to-host communication services for applications. They provide services such as connection-oriented communication, reliability, flow control, and multiplexing.

INET currently provides support for the TCP, UDP, SCTP and RTP transport layer protocols. INET nodes like `StandardHost` contain optional and replaceable instances of these protocols, like this:

```
tcp: <default("Tcp")> like ITcp if hasTcp;
udp: <default("Udp")> like IUdp if hasUdp;
sctp: <default("Sctp")> like ISctp if hasSctp;
```

As RTP is more specialized than the other ones (multimedia streaming), INET provides a separate node type, `RtpHost`, for modeling RTP traffic.

7.2 TCP

7.2.1 Overview

The TCP protocol is the most widely used protocol of the Internet. It provides reliable, ordered delivery of a stream of bytes from one application on one computer to another application on another computer. The baseline TCP protocol is described in RFC793, but other tens of RFCs contain modifications and extensions to TCP. As a result, TCP is a complex protocol and sometimes it is hard to see how the different requirements interact with each other.

There are three implementations of the TCP protocol in INET:

- `Tcp` is the primary implementation, designed for readability, extensibility, and experimentation.
- `TcpLwip` is a wrapper around the lwIP (Lightweight IP) library, a widely used open source TCP/IP stack designed for embedded systems.
- `TcpNsc` wraps Network Simulation Cradle (NSC), a library that allows real world TCP/IP network stacks to be used inside a network simulator.

All three module types implement the `ITcp` interface and communicate with other layers through the same interface, so they can be interchanged and also mixed in the same network.

It is important to note that in all three implementations, connections can be opened in either "autoread" or "explicit-read" mode, usually controlled by a boolean parameter called `autoRead`. In "autoread" mode, all incoming data in a TCP connection is immediately sent to the application, whereas in "explicit-read" mode, the user must send a specific read request similar to the Unix Socket API read() call, allowing TCP to respond with either queued data or new incoming data based on the connection.

Although read mode appears to be an implementation detail, it has fundamental implications on the behavior of the TCP protocol. "Autoread" mode always advertises the full buffer size, while "explicit-read" mode manages the advertised window based on read requests. Therefore, TCP flow control is only effective in applications that open

connections in "explicit-read" mode. Most applications use "autoread" mode, which must be taken into account when designing simulation experiments involving TCP connections.

7.2.2 Tcp

The `Tcp` simple module is the main implementation of the TCP protocol in the INET framework.

`Tcp` implements the following:

- TCP state machine
- Selection of the initial sequence number according to the system clock.
- Window-based flow control
- Window Scale option
- Persistence timer
- Keepalive timer
- Transmission policies
- RTT measurement for retransmission timeout (RTO) computation
- Delayed ACK algorithm
- Nagle's algorithm
- Silly window avoidance
- Timestamp option
- Congestion control schemes: Tahoe, Reno, New Reno, Westwood, Vegas, etc.
- Slow Start and Congestion Avoidance
- Fast Retransmit and Fast Recovery
- Loss Recovery Using Limited Transmit
- Selective Acknowledgments (SACK)
- SACK based loss recovery

Several protocol features can be turned on/off with parameters like `delayedAcksEnabled`, `nagleEnabled`, `limitedTransmitEnabled`, `increasedIWEnabled`, `sackSupport`, `windowScalingSupport`, or `timestampSupport`.

The congestion control algorithm can be selected with the `tcpAlgorithmClass` parameter. For example, the following ini file fragment selects TCP Vegas:

```
**.tcp.tcpAlgorithmClass = "TcpVegas"
```

Values like `"TcpVegas"` name C++ classes. Indeed, `Tcp` can be extended with new congestion control schemes by implementing and registering them in C++.

7.2.3 TcpLwip

lwIP is a light-weight implementation of the TCP/IP protocol suite originally written by Adam Dunkels of the Swedish Institute of Computer Science. The current development homepage is http://savannah.nongnu.org/projects/lwip/.

The implementation targets embedded devices: it has very limited resource usage (it works "with tens of kilobytes of RAM and around 40 kilobytes of ROM") and does not require an underlying OS.

The `TcpLwip` simple module is based on the 1.3.2 version of the lwIP sources.

Features:

- delayed ACK

- Nagle's algorithm
- round trip time estimation
- adaptive retransmission timeout
- SWS avoidance
- slow start threshold
- fast retransmit
- fast recovery
- persist timer
- keep-alive timer

Limitations

- only MSS and TS TCP options are supported. The TS option is turned off by default but can be enabled by defining LWIP_TCP_TIMESTAMPS to 1 in `lwipopts.h`.
- `fork` must be `true` in the passive open command
- The status request command (TCP_C_STATUS) only reports the local and remote addresses/ports of the connection and the MSS, SND.NXT, SND.WND, SND.WL1, SND.WL2, RCV.NXT, RCV.WND variables.

7.2.4 TcpNsc

Network Simulation Cradle (NSC) is a tool that allows real-world TCP/IP network stacks to be used in simulated networks. The NSC project is created by Sam Jansen and available on http://research.wand.net.nz/software/nsc.php. NSC currently contains Linux, FreeBSD, OpenBSD, and lwIP network stacks. However, on 64-bit systems, only Linux implementations can be built.

To use the `TcpNsc` module, you should download the `nsc-0.5.2.tar.bz2` package and follow the instructions in the `<inet_root>/3rdparty/README` file to build it.

Warning

Before generating the INET module, check that the `opp_makemake` call in the make file (`<inet_root>/Makefile`) includes the `-DWITH_TCP_NSC` argument. Without this option, the `TcpNsc` module is not built. If you build the INET library from the IDE, it is enough to enable the *TCP (NSC)* project feature.

Parameters

The module has the following parameters:

- `stackName`: the name of the TCP implementation to be used. Possible values are: `liblinux2.6.10.so`, `liblinux2.6.18.so`, `liblinux2.6.26.so`, `libopenbsd3.5.so`, `libfreebsd5.3.so`, and `liblwip.so`. (On the 64 bit systems, the `liblinux2.6.26.so` and `liblinux2.6.16.so` are available only).
- `stackBufferSize`: the size of the receive and send buffer of one connection for the selected TCP implementation. The NSC sets the `wmem_max`, `rmem_max`, `tcp_rmem`, `tcp_wmem` parameters to this value on Linux TCP implementations. For details, you can see the NSC documentation.

Limitations

- Because the kernel code is not reentrant, NSC creates a record containing the global variables of the stack implementation. By default, there is room for 50 instances in this table, so you cannot create more than 50 instances of `TcpNsc`. You can increase the `NUM_STACKS` constant in `num_stacks.h` and recompile NSC to overcome this limitation.
- The `TcpNsc` module does not support TCP_TRANSFER_OBJECT data transfer mode.
- The MTU of the network stack fixed to 1500, therefore MSS is 1460.

- TCP_C_STATUS command reports only local/remote addresses/ports and the current window of the connection.

7.3 UDP

The UDP protocol is a very simple datagram transport protocol, which basically provides the services of the network layer to the applications. It performs packet multiplexing and demultiplexing to ports and performs basic error detection only.

The `Udp` simple module implements the UDP protocol. There is a module interface (`IUdp`) that defines the gates of the `Udp` component. In the `StandardHost` node, the UDP component can be any module implementing that interface.

7.4 SCTP

The `Sctp` module implements the Stream Control Transmission Protocol (SCTP). Like TCP, SCTP provides reliable ordered data delivery over an unreliable network. The most prominent feature of SCTP is the capability of transmitting multiple streams of data at the same time between two end points that have established a connection.

7.5 RTP

The Real-time Transport Protocol (RTP) is a transport layer protocol for delivering audio and video over IP networks. RTP is used extensively in communication and entertainment systems that involve streaming media, such as telephony, video teleconference applications including WebRTC, television services, and web-based push-to-talk features.

The RTP Control Protocol (RTCP) is a sister protocol of the Real-time Transport Protocol (RTP). RTCP provides out-of-band statistics and control information for an RTP session.

INET provides the following modules:

- `Rtp` implements the RTP protocol
- `Rtcp` implements the RTCP protocol

CHAPTER

EIGHT

THE IPV4 PROTOCOL FAMILY

8.1 Overview

The IP protocol is the workhorse protocol of the TCP/IP protocol suite. All UDP, TCP, and ICMP packets are encapsulated into IP datagrams and transported by the IP layer. While higher-layer protocols transfer data between two communication end-points, the IP layer provides a hop-by-hop, unreliable, and connectionless delivery service. IP does not maintain any state information about the individual datagrams; each datagram is handled independently.

The nodes that are connected to the Internet can be either a host or a router. The hosts can send and receive IP datagrams, and their operating system implements the full TCP/IP stack, including the transport layer. On the other hand, routers have more than one interface card and perform packet routing between the connected networks. Routers do not need the transport layer; they work at the IP level only. The division between routers and hosts is not strict because if a host has several interfaces, they can usually be configured to operate as a router too.

Each node on the Internet has a unique IP address. IP datagrams contain the IP address of the destination. The task of the routers is to find out the IP address of the next hop on the local network and forward the packet to it. Sometimes the datagram is larger than the maximum datagram that can be sent on the link (e.g., Ethernet has a 1500-byte limit). In this case, the datagram is split into fragments and each fragment is transmitted independently. The destination host must collect all fragments and assemble the datagram before sending up the data to the transport layer.

The INET framework contains several modules to build the IPv4 network layer of hosts and routers:

- `Ipv4` is the main module that implements RFC791. This module performs IP encapsulation/decapsulation, fragmentation and assembly, and routing of IP datagrams.
- The `Ipv4RoutingTable` is a helper module that manages the routing table of the node. It is queried by the `Ipv4` module for best routes and updated by the routing daemons implementing RIP, OSPF, Manet, etc. protocols.
- The `Icmp` module can be used to generate ICMP error packets. It also supports ICMP echo applications.
- The `Arp` module performs the dynamic translation of IP addresses to MAC addresses.
- The `Igmpv2` module generates and processes multicast group membership reports.

These modules are assembled into a complete network layer module called `Ipv4NetworkLayer`. The `Ipv4NetworkLayer` module is present in `StandardHost` and `Router`.

The subsequent sections describe the IPv4 modules in detail.

8.2 IPv4

The IPv4 protocol is implemented by the `Ipv4` module.

Its parameters include:

- `checksumMode` TODO: @enum("declared", "computed") = default("declared");
- `procDelay` processing time of each incoming datagram.
- `timeToLive` default TTL of unicast datagrams.

- `multicastTimeToLive` default TTL of multicast datagrams.
- `fragmentTimeout` the maximum duration until fragments are kept in the fragment buffer.
- `limitedBroadcast` if `true`, then link-local broadcast datagrams are sent out through each interface if the higher layer did not specify the outgoing interface.
- `useProxyARP` TODO: default(true);

8.3 IPv4 Routing Table

IPv4 route tables are represented with the `Ipv4RoutingTable` module. Hosts and routers normally contain one instance of this module. The `Ipv4RoutingTable` module does not send or receive messages. Instead, C++ methods are for querying and updating the table, as well as for unicast and multicast routing.

The `Ipv4RoutingTable` module has the following parameters:

- `routerId`: for routers, the router id using IPv4 address dotted notation; specify "auto" to select the highest interface address; should be left empty for hosts.
- `forwarding`: turns IP forwarding on/off. It is always `true` in a `Router` and is `false` by default in a `StandardHost`.
- `multicastForwarding`: turns multicast IP forwarding on/off. Default is `false`, should be set to `true` in multicast routers.

The preferred method for static initialization of routing tables is to use `Ipv4NetworkConfigurator`. While `Ipv4RoutingTable` can read the routes from a *routing file*, that is considered obsolete. Old routing files should be replaced with the XML configuration of `Ipv4NetworkConfigurator`. The *Network Autoconfiguration* chapter describes the format of the new configuration files.

8.4 ICMP

The Internet Control Message Protocol (ICMP) can be modeled with the `Icmp` module. ICMP is the error reporting and diagnostic mechanism of the Internet. It uses the services of IPv4, so it is a transport layer protocol, but unlike TCP or UDP, it is not used to transfer user data. It cannot be separated from IPv4, because the routing errors are reported by ICMP.

The `Icmp` module can be used to send error messages and ping requests. It can also respond to incoming ICMP messages.

Each ICMP message is encapsulated within an IP datagram, so its delivery is unreliable.

8.5 ARP

The Address Resolution Protocol (ARP) is modeled with the `Arp` module. The ARP protocol is designed to translate a local protocol address to a hardware address. Although the ARP protocol can be used with several network protocols and hardware addressing schemes, in practice, they are almost always IPv4 and 802.3 addresses. The `Arp` module only supports IPv4-to-MAC address translation, but not the opposite direction, Reverse ARP (RARP).

The address to be resolved can be either an IPv4 broadcast/multicast or a unicast address. The corresponding MAC addresses can be computed for broadcast and multicast addresses (RFC 1122, 6.4); unicast addresses are resolved using the ARP protocol.

If the MAC address is found in the ARP cache, then the packet is transmitted to the addressed interface immediately. Otherwise, the packet is queued and an address resolution takes place.

For address resolution, ARP broadcasts a request frame on the network. In the request, it publishes its own IP and MAC addresses so each node in the local subnet can update their mapping. The node whose MAC address was requested will respond with an ARP frame containing its own MAC address directly to the node that sent the request. When the original node receives the ARP response, it updates its ARP cache and sends the delayed IP packet using the learned MAC address.

ARP resolution is initiated with a C++ call.

The module parameters of `Arp` are:

- `retryTimeout`: number of seconds ARP waits between retries to resolve an IPv4 address (default is 1s)
- `retryCount`: number of times ARP will attempt to resolve an IPv4 address (default is 3)
- `cacheTimeout`: number of seconds unused entries in the cache will time out (default is 120s)
- `proxyARP`: enables proxy ARP mode (default is `true`)
- `globalARP`: use global ARP cache (default is `false`)

8.6 IGMP

The `Igmpv3` module implements the Internet Group Management Protocol (IGMP). IGMP is a communications protocol used by hosts and adjacent routers on IPv4 networks to establish multicast group memberships. IGMP is an integral part of IP multicast.

IGMP is responsible for distributing the information of multicast group memberships from hosts to routers. When an interface of a host joins a multicast group, it will send an IGMP report on that interface to routers. It can also send reports when the interface leaves the multicast group, so it does not want to receive those multicast datagrams. The IGMP module of multicast routers processes these IGMP reports: it updates the list of groups that have members on the link of the incoming message.

The `IIgmp` module interface defines the connections of IGMP modules. IGMP reports are transmitted by IP, so the module contains gates to be connected to the IP module (`ipIn/ipOut`). The IP module delivers packets with protocol number 2 to the IGMP module. However, some multicast routing protocols (like DVMRP) also exchange routing information by sending IGMP messages, so they should be connected to the `routerIn/routerOut` gates of the IGMP module. The IGMP module delivers the IGMP messages not processed by itself to the connected routing module.

The `Igmpv2` module implements version 2 of the IGMP protocol (RFC 2236). Next, we describe its behavior in hosts and routers in detail. Note that multicast routers behave as hosts too, i.e. they are sending reports to other routers when joining or leaving a multicast group.

8.6.1 Host behavior

When an interface joins a multicast group, the host will send a Membership Report immediately to the group address. This report is repeated after `unsolicitedReportInterval` to cover the possibility of the first report being lost.

When a host's interface leaves a multicast group, and it was the last host that sent a Membership Report for that group, it will send a Leave Group message to the all-routers multicast group (224.0.0.2).

This module also responds to IGMP Queries. When the host receives a Group-Specific Query on an interface that belongs to that group, then it will set a timer to a random value between 0 and Max Response Time of the Query. If the timer expires before the host observes a Membership Report sent by other hosts, then the host sends an IGMPv2 Membership Report. When the host receives a General Query on an interface, a timer is initialized and a report is sent for each group membership of the interface.

8.6.2 Router behavior

Multicast routers maintain a list for each interface containing the multicast groups that have listeners on that interface. This list is updated when IGMP Membership Reports and Leave Group messages arrive, or when a timer expires since the last Query.

When multiple routers are connected to the same link, the one with the smallest IP address will be the Querier. When other routers observe that they are Non-Queriers (by receiving an IGMP Query with a lower source address), they stop sending IGMP Queries until `otherQuerierPresentInterval` has elapsed since the last received query.

Routers periodically (`queryInterval`) send a General Query on each attached network for which this router is a Querier. On startup, the router sends `startupQueryCount` queries separated by

startupQueryInterval. A General Query has an unspecified Group Address field, a Max Response Time field set to queryResponseInterval, and is sent to the all-systems multicast address (224.0.0.1).

When a router receives a Membership Report, it will add the reported group to the list of multicast group memberships. At the same time, it will set a timer for the membership to groupMembershipInterval. Repeated reports restart the timer. If the timer expires, the router assumes that the group has no local members, and multicast traffic is no longer forwarded to that interface.

When a Querier receives a Leave Group message for a group, it sends a Group-Specific Query to the group being left. It repeats the Query lastMemberQueryCount times separated by lastMemberQueryInterval until a Membership Report is received. If no Report is received, then the router assumes that the group has no local members.

8.6.3 Parameters

The following parameters have effects in both hosts and routers:

- enabled if false, then the IGMP module never sends any messages and discards incoming messages. The default is true.

The following parameters are only used in hosts:

- unsolicitedReportInterval the time between repetitions of a host's initial report of membership in a group. The default is 10s.

Router timeouts are configured by these parameters:

- robustnessVariable the IGMP is robust to robustnessVariable-1 packet losses. The default is 2.
- queryInterval the interval between General Queries sent by a Querier. The default is 125s.
- queryResponseInterval the Max Response Time inserted into General Queries.
- groupMembershipInterval the amount of time that must pass before a multicast router decides there are no more members of a group on a network. Fixed to robustnessVariable * queryInterval + queryResponseInterval.
- otherQuerierPresentInterval the length of time that must pass before a multicast router decides that there is no longer another multicast router which should be the querier. Fixed to robustnessVariable * queryInterval + queryResponseInterval / 2.
- startupQueryInterval the interval between General Queries sent by a Querier on startup. The default is queryInterval / 4.
- startupQueryCount the number of Queries sent out on startup, separated by the startupQueryInterval. The default is robustnessVariable.
- lastMemberQueryInterval the Max Response Time inserted into Group-Specific Queries sent in response to Leave Group messages, and is also the amount of time between Group-Specific Query messages. The default is 1s.
- lastMemberQueryCount the number of Group-Specific Queries sent before the router assumes there are no local members. The default is robustnessVariable.

CHAPTER

NINE

THE IPV6 PROTOCOL FAMILY

9.1 Overview

IPv6 is the successor of IPv4, designed primarily to overcome the exhaustion of the IPv4 address space. It uses 128-bit addresses (written as colon-separated groups of hexadecimal digits, e.g. `2001:db8::1`), and refines the protocol in several ways: address autoconfiguration, a simplified and fixed-size base header with optional extension headers, routers that never fragment (the source performs Path MTU Discovery instead), and the replacement of ARP, ICMP Router Discovery and ICMP Redirect with a single Neighbor Discovery protocol built on ICMPv6.

Like IPv4, the IPv6 network layer is built from several cooperating modules:

- `Ipv6` is the main module that implements RFC 8200. It performs encapsulation/decapsulation, routing and forwarding of datagrams, fragmentation and reassembly, and the processing of extension headers.
- `Ipv6RoutingTable` is a helper module that holds the routing table together with the Neighbor Discovery data structures (the destination cache, neighbor cache and prefix list). It is queried and updated by the other modules through C++ method calls and does not send or receive messages.
- `Icmpv6` implements ICMPv6 error messages and the Echo (ping) service.
- `Ipv6NeighbourDiscovery` implements Neighbor Discovery (RFC 4861) and stateless address autoconfiguration (RFC 4862): Router and Neighbor Solicitation/Advertisement, Duplicate Address Detection, and Neighbor Unreachability Detection.

These modules are assembled into a complete network layer module, `Ipv6NetworkLayer`, which is present in `StandardHost6` and `Router6`.

The network layer can optionally include Mobile IPv6 (RFC 3775); see the *Mobile IPv6* section. The `Ipv6NetworkConfigurator` module assigns addresses and routes; see the *Network Autoconfiguration* chapter.

The subsequent sections describe the IPv6 modules in detail.

9.2 IPv6

The IPv6 protocol is implemented by the `Ipv6` module. It accepts packets from higher-layer protocols (which attach an `L3AddressReq` to specify the destination), encapsulates them, and routes them by querying the `Ipv6RoutingTable` in C++. Received datagrams are reassembled if fragmented, their extension headers are processed, and the payload is delivered to the registered upper-layer protocol selected by the header's protocol field (with an `L3AddressInd` attached).

Other modules may register *netfilter hooks* on `Ipv6` to inspect, modify, drop or queue datagrams at well-defined points (pre-routing, local-out, post-routing). Mobile IPv6 uses these hooks, so the IPv6 core itself contains no mobility-specific code.

Its parameters include:

- `procDelay`: processing time of each incoming datagram (default 0s).

9.3 IPv6 Routing Table

The IPv6 routing table is represented by the `Ipv6RoutingTable` module. Hosts and routers normally contain one instance. Like its IPv4 counterpart, it does not send or receive messages; C++ methods are used for querying and updating the routes, the destination cache and the neighbor/prefix caches.

Its parameters include:

- `forwarding`: turns IP forwarding on/off. It is `true` in a `Router6` and `false` by default in a `StandardHost6`.
- `multicastForwarding`: turns multicast forwarding on/off (default `false`).
- Router Advertisement defaults applied to all router interfaces, such as `advManagedFlag`, `advOtherConfigFlag`, `advLinkMtu`, `advCurHopLimit` and `advDefaultLifetime`.

The preferred way to set up addresses and routes is the `Ipv6NetworkConfigurator`, described in the *Network Autoconfiguration* chapter. The module can also read manual routes from its `routes` XML parameter; a <route> element accepts the destination prefix either as a separate `prefixLength` attribute or in CIDR `address/prefixlen` notation.

9.4 ICMPv6

ICMPv6 (RFC 4443) is modeled by the `Icmpv6` module. It is the error reporting and diagnostic mechanism of IPv6: it generates and processes Destination Unreachable, Packet Too Big, Time Exceeded and Parameter Problem messages on behalf of `Ipv6`, and answers Echo Requests (ping) with Echo Replies. Each ICMPv6 message is carried in an IPv6 datagram, so its delivery is unreliable.

Unlike in IPv4, the Neighbor Discovery messages (Router/Neighbor Solicitation and Advertisement, Redirect) are *not* handled here but by the `Ipv6NeighbourDiscovery` module, even though they too are ICMPv6 message types.

9.5 Neighbor Discovery and Autoconfiguration

The `Ipv6NeighbourDiscovery` module implements IPv6 Neighbor Discovery (RFC 4861) and stateless address autoconfiguration (SLAAC, RFC 4862). It subsumes several functions that are separate protocols in IPv4 (ARP, ICMP Router Discovery, ICMP Redirect):

- *Address autoconfiguration*: on startup, an interface forms a link-local address from its interface identifier, and — on hosts — derives global addresses from the prefixes advertised by routers.
- *Router discovery*: routers periodically multicast Router Advertisements (and answer Router Solicitations) carrying on-link prefixes, the default-router lifetime, MTU and other parameters. Hosts use them to configure addresses and a default route.
- *Address resolution*: a node resolves a neighbor's link-layer address by multicasting a Neighbor Solicitation and receiving a Neighbor Advertisement (the IPv6 equivalent of ARP).
- *Duplicate Address Detection (DAD)*: before an autoconfigured address is used, the node verifies it is unique on the link by soliciting it.
- *Neighbor Unreachability Detection (NUD)*: a node tracks whether a neighbor is still reachable and re-resolves it when needed.

The Neighbor Discovery data structures (neighbor cache, destination cache, prefix list, default router list) live in the `Ipv6RoutingTable`.

Its parameters include the Router Advertisement timing (`minIntervalBetweenRAs`, `maxIntervalBetweenRAs`, `advReachableTime`), the number of DAD solicitations (`dupAddrDetectTransmits`, `0` disables DAD), optimistic DAD (`optimisticDad`, RFC 4429), and retransmission timers (`retransTimer`, `baseReachableTime`).

9.6 IPv6 Tunneling

IPv6-in-IPv6 tunneling (RFC 2473) is modeled as an ordinary virtual network interface, `Ipv6TunnelInterface`. Traffic routed to such an interface is encapsulated in an outer IPv6 header (from the tunnel entry point toward the tunnel exit point) and handed back to `Ipv6`, which forwards the encapsulated datagram like any other packet; at the exit point it is decapsulated and the inner datagram re-enters the IPv6 layer as ordinary ingress. A tunnel is therefore just *an interface plus a route*, and the IPv6 core needs no tunneling-specific code. Mobile IPv6 uses these interfaces for its home-agent and reverse tunnels.

9.7 Mobile IPv6

Mobile IPv6 (MIPv6, RFC 3775) lets a node remain reachable under a stable address while it moves around the IPv6 Internet. A *mobile node* (MN) owns a permanent **home address** on its home link, and acquires a **care-of address** at each new point of attachment. Three roles participate:

- the **mobile node** (MN), which moves and registers its current care-of address;
- the **home agent** (HA), a router on the home link that intercepts traffic addressed to the MN's home address and tunnels it to the MN's care-of address;
- the **correspondent node** (CN), any peer the MN communicates with, which may additionally support *route optimization*.

The protocol logic is implemented by the `Mipv6` module, a submodule of `Ipv6NetworkLayer` that is present when its `hasMipv6` parameter is set. The role is selected by the `isMobileNode` and `isHomeAgent` parameters of `Mipv6`. Two passive data-store modules accompany it: the `BindingUpdateList` (on a mobile node, recording the bindings it has registered with its home agent and correspondent nodes) and the `BindingCache` (on a home agent or correspondent node, recording the bindings registered by mobile nodes).

These pieces are pre-wired in the ready-made node types: `MobileHost6` and the wireless `WirelessHost6` act as mobile nodes, `HomeAgent6` as a home agent, and `CorrespondentNode6` as a route-optimization-capable correspondent.

The operation proceeds as follows:

- *Home registration and bidirectional tunneling*: when the MN detects (via Neighbor Discovery) that it has moved to a foreign link, it forms a care-of address and sends a Binding Update to its home agent. The HA records the binding in its `BindingCache` and, from then on, intercepts packets addressed to the MN's home address and tunnels them to its care-of address; the MN reverse-tunnels its replies through the HA. The tunnels are `Ipv6TunnelInterface` interfaces.
- *Route optimization*: to avoid the triangular path through the home agent, the MN can register its care-of address directly with a correspondent node. First the *return-routability* procedure (Home Test Init/Care-of Test Init and the corresponding Home Test/Care-of Test messages) proves the MN owns both addresses; then the MN sends a Binding Update to the CN. Afterwards the CN sends directly to the care-of address using a Type 2 Routing Header, and the MN sends to the CN using a Home Address Option — both inserted by `Mipv6` through netfilter hooks, again without special-casing the IPv6 core.
- *Handover and return home*: on each move the MN re-registers its new care-of address (and refreshes the route-optimized bindings); when it returns to its home link it de-registers, tearing down the tunnels.

Binding lifetimes are bounded by the `maxHaBindingLifeTime` (home registration) and `maxRrBindingLifeTime` (route optimization) parameters of `Mipv6`; bindings are refreshed before they expire.

A complete, runnable example is provided under `examples/ipv6/mipv6`. Note that Mobile IPv6 (network-layer mobility) is distinct from the physical mobility models described in the *Node Mobility* chapter, although a wireless mobile node typically uses both.

CHAPTER

TEN

OTHER NETWORK PROTOCOLS

10.1 Overview

Network layer protocols in INET are not restricted to IPv4 and IPv6. INET nodes such as `Router` and `StandardHost` can be configured to use alternative network layer protocols instead of, or in addition to, IPv4 and IPv6.

Node models contain three optional network layers that can be individually turned on or off:

```
ipv4: <default("Ipv4NetworkLayer")> like INetworkLayer if hasIpv4;
ipv6: <default("Ipv6NetworkLayer")> like INetworkLayer if hasIpv6;
generic: <default("")> like INetworkLayer if hasGn;
```

In the default configuration, only IPv4 is turned on. If an alternative network layer protocol is desired instead of IPv4/IPv6, the configuration should look something like this:

```
**.hasIpv4 = false
**.hasIpv6 = false
**.hasGn = true
**.generic.typename = "WiseRouteNetworkLayer"
```

The list of alternative network layers includes:

- `SimpleNetworkLayer` is a generic network layer where the concrete protocol type is a parameter
- `NextHopNetworkLayer` is a network layer specialized for the "Next-Hop Forwarding Protocol", an abstract implementation of the next-hop routing concept
- `WiseRouteNetworkLayer` is specialized for the Wise Route protocol

The list of network layer protocols that can be plugged into `SimpleNetworkLayer` includes:

- `Flooding` implements controlled flooding
- `WiseRoute` implements Wise Route, a convergecasting protocol for wireless sensor networks
- `ProbabilisticBroadcast` implements a multi-hop ad-hoc data dissemination protocol
- `AdaptiveProbabilisticBroadcast` is a variant of the previous one

In addition to the network layer protocol, `SimpleNetworkLayer` includes an instance of `GlobalArp` for address resolution, and an instance of `EchoProtocol`, a module type that implements a simple *ping*-like protocol.

All the above network protocols can work with IPv4 addresses, IPv6 addresses, use MAC address as a network address (this is sometimes useful in WSNs), or use synthetic addresses that only make sense within the simulation.[1]

In apps, the network layer protocol to be used needs to be specified. For example:

```
**.app[*].networkProtocol = "flood"
```

[1] This is possible because the implementation of these modules simply uses the `L3Address` C++ class, which is a variant type capable of holding several types of L3 addresses.

10.2 Protocols

10.2.1 Flooding

`Flooding` is a simple flooding protocol for network-level broadcast. Already broadcast messages are remembered and not rebroadcasted if another copy of that message is received.

10.2.2 ProbabilisticBroadcast

`ProbabilisticBroadcast` is a multi-hop ad-hoc data dissemination protocol based on probabilistic broadcast.

The number of packets sent on the channel is reduced (reducing the broadcast storm problem) at the risk of some nodes not receiving the data. It is particularly interesting for mobile networks.

The transmission probability for each attempt, the time between two transmission attempts, the maximum number of broadcast transmissions of a packet, and some other settings are parameters.

10.2.3 AdaptiveProbabilisticBroadcast

`AdaptiveProbabilisticBroadcast` is a variant of `ProbabilisticBroadcast` that automatically adjusts transmission probabilities depending on the estimated number of neighbors.

10.2.4 WiseRoute

`WiseRoute` implements Wise Route, a simple loop-free routing algorithm that builds a routing tree from a central network point, designed for sensor networks and convergecast traffic (Wireless Sensor routing).

A route building message is broadcasted by the sink (the device at the center of the network). Each network node that receives it selects the sink as a parent in the routing tree and rebroadcasts the route building message. This procedure maximizes the probability that all network nodes can join the network and avoids loops.

The `sinkAddress` parameter specifies the sink network address, `rssiThreshold` is a threshold to avoid using bad links (with too low RSSI values) for routing, and `routeFloodsInterval` should be set to zero for all nodes except the sink. Each `routeFloodsInterval`, the sink restarts the tree building procedure. Set it to a large value if the tree should not be rebuilt.

10.2.5 NextHopForwarding

The `NextHopForwarding` module is an implementation of the next-hop forwarding concept. (It can be thought of as an abstracted version of IP.)

The protocol needs an additional module, a `NextHopRoutingTable`, for its operation. The routing table module is included in the `NextHopNetworkLayer` compound module.

10.3 Address Types

The following address types are available:

- IPv4 address
- IPv6 address
- MAC address
- module ID
- module path

Protocols described in this chapter work with "generic" L3 addresses; they can use any address type.

When choosing IPv4 addresses, an `Ipv4NetworkConfigurator` global instance can be used to assign addresses to network interfaces. (Note that `Ipv4NetworkConfigurator` also needs a per-node instance of `Ipv4NodeConfigurator` for it to work.)

10.4 Address Resolution

Address resolution is done by `GlobalArp`. If the address type is IPv4, `Arp` can be used instead of `GlobalArp`.

CHAPTER

ELEVEN

NETWORK AUTOCONFIGURATION

11.1 Overview

This chapter describes static autoconfiguration of networks: assigning addresses and filling in routing tables at the start of the simulation. It covers the IPv4 and IPv6 network configurators and layer-2 configuration.

11.2 Configuring IPv4 Networks

An IPv4 network is composed of several nodes like hosts, routers, switches, hubs, Ethernet buses, or wireless access points. The nodes having an IPv4 network layer (hosts and routers) should be configured at the beginning of the simulation. The configuration assigns IP addresses to the nodes and fills their routing tables. If multicast forwarding is simulated, then the multicast routing tables also must be filled in.

The configuration can be manual (each address and route is fully specified by the user) or automatic (addresses and routes are generated by a configurator module at startup).

Before version 1.99.4, INET offered `Ipv4FlatNetworkConfigurator` for automatic, and routing files for manual configuration. Both configuration methods had serious limitations, so a new configurator has been added in version 1.99.4: `Ipv4NetworkConfigurator`. This configurator supports both fully manual and fully automatic configuration. It can also be used with partially specified manual configurations; the configurator fills in the gaps automatically.

The next section describes the usage of `Ipv4NetworkConfigurator`. The legacy solutions `Ipv4FlatNetworkConfigurator` and routing files are described in subsequent sections.

11.2.1 Ipv4NetworkConfigurator

The `Ipv4NetworkConfigurator` assigns IP addresses and sets up static routing for an IPv4 network.

It assigns per-interface IP addresses, strives to take subnets into account, and can also optimize the generated routing tables by merging routing entries.

Hierarchical routing can be set up by using only a fraction of configuration entries compared to the number of nodes. The configurator also does routing table optimization that significantly decreases the size of routing tables in large networks.

The configuration is performed in stage 2 of the initialization. At this point, interface modules (e.g. PPP) have already registered their interfaces in the interface table. If an interface is named `ppp[0]`, then the corresponding interface entry is named `ppp0`. This name can be used in the config file to refer to the interface.

The configurator goes through the following steps:

1. Builds a graph representing the network topology. The graph will have a vertex for every module that has a `@node` property (this includes hosts, routers, and L2 devices like switches, access points, Ethernet hubs, etc.). It also assigns weights to vertices and edges that will be used by the shortest path algorithm when setting up routes. Weights will be infinite for IP nodes that have IP forwarding disabled (to prevent routes from transiting them) and zero for all other nodes (routers and L2 devices). Edge weights are chosen to be inversely proportional to the bitrate of the link so that the configurator prefers connections with higher

bandwidth. For internal purposes, the configurator also builds a table of all "links" (the link data structure consists of the set of network interfaces that are on the same point-to-point link or LAN).

2. Assigns IP addresses to all interfaces of all nodes. The assignment process takes into consideration the addresses and netmasks already present on the interfaces (possibly set in earlier initialization stages) and the configuration provided in the XML format (described below). The configuration can specify "templates" for the address and netmask, with parts that are fixed and parts that can be chosen by the configurator (e.g. "10.0.x.x"). In the most general case, the configurator is allowed to choose any address and netmask for all interfaces (which results in automatic address assignment). In the most constrained case, the configurator is forced to use the requested addresses and netmasks for all interfaces (which translates to manual address assignment). There are many possible configuration options between these two extremes. The configurator assigns addresses in a way that maximizes the number of nodes per subnet. Once it figures out the nodes that belong to a single subnet, it will optimize for allocating the longest possible netmask. The configurator might fail to assign netmasks and addresses according to the given configuration parameters; if that happens, the assignment process stops, and an error is signaled.
3. Adds the manual routes that are specified in the configuration.
4. Adds static routes to all routing tables in the network. The configurator uses Dijkstra's weighted shortest path algorithm to find the desired routes between all possible node pairs. Then it will populate the routing tables with entries to allow accessing all destination Interfaces in the network. The configurator can be safely instructed to add default routes where applicable, significantly reducing the size of the host routing tables. It can also add subnet routes instead of interface routes, further reducing the size of routing tables. Turning on this option requires careful design to avoid having IP addresses from the same subnet on different links. CAVEAT: Using manual routes and static route generation together may have unwanted side effects because route generation ignores manual routes.
5. Then it optimizes the routing tables for size. This optimization allows configuring larger networks with a smaller memory footprint and makes the routing table lookup faster. The resulting routing table might be different in that it will route packets that the original routing table did not. Nevertheless, the following invariant holds: any packet routed by the original routing table (has matching route) will still be routed the same way by the optimized routing table.
6. Finally, it dumps the requested results of the configuration. It can dump the network topology, assigned IP addresses, routing tables, and its own configuration format.

The module can dump the result of the configuration in the XML format, which it can read. This is useful to save the result of a time-consuming configuration (large network with optimized routes) and use it as the config file of subsequent runs.

Network topology graph

The network topology graph is constructed from the nodes of the network. The node is a module having a `@node` property (this includes hosts, routers, and L2 devices like switches, access points, Ethernet hubs, etc.). An IP node is a node that contains an `InterfaceTable` and an `Ipv4RoutingTable`. A router is an IP node that has multiple network interfaces, and IP forwarding is enabled in its routing table module. In multicast routers, the `forwardMulticast` parameter is also set to `true`.

A link is a set of interfaces that can send datagrams to each other without intervening routers. Each interface belongs to exactly one link. For example, two interfaces connected by a point-to-point connection form a link. Ethernet interfaces connected via buses, hubs, or switches. The configurator identifies links by discovering the connections between the IP nodes, buses, hubs, and switches.

Wireless links are identified by the `ssid` or `accessPointAddress` parameter of the 802.11 management module. Wireless interfaces whose node does not contain a management module are supposed to be on the same wireless link. Wireless links can also be configured in the configuration file of `Ipv4NetworkConfigurator`:

```
<config>
  <wireless hosts="area1.*" interfaces="wlan*"/>
</config>
```

This puts WLAN interfaces of the specified hosts into the same wireless link.

If a link contains only one router, it is marked as the gateway of the link. Each datagram whose destination is outside the link must go through the gateway.

Address assignment

Addresses can be set up manually by giving the address and netmask for each IP node. If some part of the address or netmask is unspecified, then the configurator can fill them automatically. Unspecified fields are given as an "x" character in the dotted notation of the address. For example, if the address is specified as 192.168.1.1 and the netmask is 255.255.255.0, then the node address will be 192.168.1.1, and its subnet is 192.168.1.0. If it is given as 192.168.x.x and 255.255.x.x, then the configurator chooses a subnet address in the range of 192.168.0.0 - 192.168.255.252 and an IP address within the chosen subnet. (The maximum subnet mask is 255.255.255.252, which allows 2 nodes in the subnet.)

The following configuration generates network addresses below the 10.0.0.0 address for each link and assigns unique IP addresses to each host:

```
<config>
  <interface hosts="*" address="10.x.x.x" netmask="255.x.x.x"/>
</config>
```

The configurator tries to put nodes on the same link into the same subnet, so it's enough to configure the address of only one node on each link.

The following example configures a hierarchical network in a way that keeps routing tables small:

```
<config>
  <interface hosts="area11.lan1.*" address="10.11.1.x" netmask="255.255.255.x"/>
  <interface hosts="area11.lan2.*" address="10.11.2.x" netmask="255.255.255.x"/>
  <interface hosts="area12.lan1.*" address="10.12.1.x" netmask="255.255.255.x"/>
  <interface hosts="area12.lan2.*" address="10.12.2.x" netmask="255.255.255.x"/>
  <interface hosts="area*.router*" address="10.x.x.x" netmask="x.x.x.x"/>
  <interface hosts="*" address="10.x.x.x" netmask="255.x.x.0"/>
</config>
```

The XML configuration must contain exactly one <config> element. Under the root element, there can be multiple of the following elements:

The interface element provides configuration parameters for one or more interfaces in the network. The selector attributes limit the scope where the interface element has effects. The parameter attributes limit the range of assignable addresses and netmasks. The <interface> element may contain the following attributes:

- @hosts is an optional selector attribute that specifies a list of host name patterns. Only interfaces in the specified hosts are affected. The pattern might be a full path starting from the network, or a module name anywhere in the hierarchy, and other patterns similar to ini file keys. The default value is "*", which matches all hosts. For example, "subnet.client*" or "host* router[0..3]" or "area*.*.host[0]".
- @names is an optional selector attribute that specifies a list of interface name patterns. Only interfaces with the specified names are affected. The default value is "*", which matches all interfaces. For example, "eth* ppp0" or "*".
- @towards is an optional selector attribute that specifies a list of host name patterns. Only interfaces connected towards the specified hosts are affected. The specified name will be matched against the names of hosts that are on the same LAN as the one that is being configured. This works even if there's a switch between the configured host and the one specified here. For wired networks, it might be easier to specify this parameter instead of specifying the interface names. The default value is "*". For example, "ap" or "server" or "client*".
- @among is an optional selector attribute that specifies a list of host name patterns. Only interfaces in the specified hosts connected towards the specified hosts are affected. The 'among="X Y Z"' is the same as 'hosts="X Y Z" towards="X Y Z"'.
- @address is an optional parameter attribute that limits the range of assignable addresses. Wildcards are allowed using the 'x' character as part of the address in place of a byte. Unspecified parts will be filled auto-

matically by the configurator. The default value is "", which means that the address will not be configured. Unconfigured interfaces still have allocated addresses in their subnets, allowing them to become configured later very easily. For example, "192.168.1.1" or "10.0.x.x".

- `@netmask` is an optional parameter attribute that limits the range of assignable netmasks. Wildcards are allowed using the 'x' character as part of the netmask in place of a byte. Unspecified parts will be filled automatically by the configurator. The default value is "", which means that any netmask can be configured. For example, "255.255.255.0" or "255.255.x.x" or "255.255.x.0".
- `@mtu` is a number optional parameter attribute to set the MTU parameter in the interface. When unspecified, the interface parameter is left unchanged.
- `@metric` is a number optional parameter attribute to set the Metric parameter in the interface. When unspecified, the interface parameter is left unchanged.

Wireless interfaces can similarly be configured by adding <wireless> elements to the configuration. Each <wireless> element with a different id defines a separate subnet.

- `@id` (optional) identifies a wireless network, a unique value used if missed.
- `@hosts` is an optional selector attribute that specifies a list of host name patterns. Only interfaces in the specified hosts are affected. The default value is "*", which matches all hosts.
- `@interfaces` is an optional selector attribute that specifies a list of interface name patterns. Only interfaces with the specified names are affected. The default value is "*", which matches all interfaces.

Multicast groups

Multicast groups can be configured by adding <multicast-group> elements to the configuration file. Interfaces that belong to a multicast group will join the group automatically.

For example:

```
<config>
  <multicast-group hosts="router*" interfaces="eth*" address="224.0.0.5"/>
</config>
```

This adds all Ethernet interfaces of nodes whose name starts with "router" to the 224.0.0.5 multicast group.

The <multicast-group> element has the following attributes:

- `@hosts` is an optional selector attribute that specifies a list of host name patterns. Only interfaces in the specified hosts are affected. The default value is "*", which matches all hosts.
- `@interfaces` is an optional selector attribute that specifies a list of interface name patterns. Only interfaces with the specified names are affected. The default value is "*", which matches all interfaces.
- `@towards` is an optional selector attribute that specifies a list of host name patterns. Only interfaces connected towards the specified hosts are affected. The default value is "*".
- `@among` is an optional selector attribute that specifies a list of host name patterns. Only interfaces in the specified hosts connected towards the specified hosts are affected. The 'among="X Y Z"' is the same as 'hosts="X Y Z" towards="X Y Z"'.
- `@address` is a mandatory parameter attribute that specifies a list of multicast group addresses to be assigned. Values must be selected from the valid range of multicast addresses. For example, "224.0.0.1 224.0.1.33".

Manual route configuration

The `Ipv4NetworkConfigurator` module allows the user to fully specify the routing tables of IP nodes at the beginning of the simulation.

The <route> elements of the configuration add a route to the routing tables of selected nodes. The element has the following attributes:

- `@hosts` is an optional selector attribute that specifies a list of host name patterns. Only routing tables in the specified hosts are affected. The default value is "", which means all hosts will be affected. For example, "host* router[0..3]".
- `@destination` is an optional parameter attribute that specifies the destination address in the route (L3AddressResolver syntax). The default value is "*", which means all destinations. For example, "192.168.1.1" or "subnet.client[3]" or "subnet.server(ipv4)" or "*".
- `@netmask` is an optional parameter attribute that specifies the netmask in the route. The default value is "*", which means all netmasks. For example, "255.255.255.0" or "/29" or "*".
- `@gateway` is an optional parameter attribute that specifies the gateway (next-hop) address in the route (L3AddressResolver syntax). When unspecified, the interface parameter must be specified. The default value is "*", which means all gateways. For example, "192.168.1.254" or "subnet.router" or "*".
- `@interface` is an optional parameter attribute that specifies the output interface name in the route. When unspecified, the gateway parameter must be specified. This parameter has no default value. For example, "eth0".
- `@metric` is an optional parameter attribute that specifies the metric in the route. The default value is 0.

Multicast routing tables can similarly be configured by adding `<multicast-route>` elements to the configuration.

- `@hosts` is an optional selector attribute that specifies a list of host name patterns. Only routing tables in the specified hosts are affected. For example, "host* router[0..3]".
- `@source` is an optional parameter attribute that specifies the address of the source network. The default value is "*", which matches all sources.
- `@netmask` is an optional parameter attribute that specifies the netmask of the source network. The default value is "*", which matches all sources.
- `@groups` is an optional list of IPv4 multicast addresses specifying the groups this entry applies to. The default value is "*", which matches all multicast groups. For example, "225.0.0.1 225.0.1.2".
- `@metric` is an optional parameter attribute that specifies the metric in the route.
- `@parent` is an optional parameter attribute that specifies the name of the interface the multicast datagrams are expected to arrive. When a datagram arrives on the parent interface, it will be forwarded towards the child interfaces; otherwise, it will be dropped. The default value is the interface on the shortest path towards the source of the datagram.
- `@children` is a mandatory parameter attribute that specifies a list of interface name patterns:
  - a name pattern (e.g. "ppp*") matches the name of the interface.
  - a 'towards' pattern (starting with ">", e.g. ">router*") matches the interface by naming one of the neighbor nodes on its link.

  Incoming multicast datagrams are forwarded to each child interface except the one they arrived in.

The following example adds an entry to the multicast routing table of `router1` that instructs the routing algorithm to forward multicast datagrams whose source is in the 10.0.1.0 network and whose destination address is 225.0.0.1 to send on the `eth1` and `eth2` interfaces assuming it arrived on the `eth0` interface:

```
<multicast-route hosts="router1" source="10.0.1.0" netmask="255.255.255.0"
                 groups="225.0.0.1" metric="10"
                 parent="eth0" children="eth1 eth2"/>
```

Automatic route configuration

If the `addStaticRoutes` parameter is true, then the configurator adds static routes to all routing tables.

The configurator uses Dijkstra's weighted shortest path algorithm to find the desired routes between all possible node pairs. As a result, each routing table will be populated with entries to allow accesssing all destination interfaces in the network directly or indirectly.

The configurator can be safely instructed to add default routes where applicable, significantly reducing the size of the host routing tables. It can also add subnet routes instead of destination interface routes, further reducing the size of routing tables. Turning on this option requires careful design to avoid having IP addresses from the same subnet on different links.

Caution

Using manual routes and static route generation together may have unwanted side effects because route generation ignores manual routes. Therefore, if the configuration file contains manual routes, then the `addStaticRoutes` parameter should be set to `false`.

Route optimization

If the `optimizeRoutes` parameter is `true` then the configurator tries to optimize the routing table for size. This optimization allows configuring larger networks with smaller memory footprint and makes the routing table lookup faster.

The optimization is performed by merging routes whose gateway and outgoing interface is the same by finding a common prefix that matches only those routes. The resulting routing table might be different in that it will route packets that the original routing table did not. Nevertheless, the following invariant holds: any packet routed by the original routing table (has matching route) will still be routed the same way by the optimized routing table.

Parameters

This list summarize the parameters of the `Ipv4NetworkConfigurator`:

- `config`: XML configuration parameters for IP address assignment and adding manual routes.
- `assignAddresses`: assign IP addresses to all interfaces in the network
- `assignDisjunctSubnetAddresses`: avoid using the same address prefix and netmask on different links when assigning IP addresses to interfaces
- `addStaticRoutes`: add static routes to the routing tables of all nodes to route to all destination interfaces (only where applicable; turn off when config file contains manual routes)
- `addDefaultRoutes`: add default routes if all routes from a source node go through the same gateway (used only if addStaticRoutes is true)
- `addSubnetRoutes`: add subnet routes instead of destination interface routes (only where applicable; used only if addStaticRoutes is true)
- `optimizeRoutes`: optimize routing tables by merging routes, the resulting routing table might route more packets than the original (used only if addStaticRoutes is true)
- `dumpTopology`: if true, then the module prints extracted network topology
- `dumpAddresses`: if true, then the module prints assigned IP addresses for all interfaces
- `dumpRoutes`: if true, then the module prints configured and optimized routing tables for all nodes to the module output
- `dumpConfig`: name of the file, write configuration into the given config file that can be fed back to speed up subsequent runs (network configurations)

11.2.2 Ipv4FlatNetworkConfigurator (Legacy)

The `Ipv4FlatNetworkConfigurator` module configures IP addresses and routes of IP nodes of a network. All assigned addresses share a common subnet prefix, and the network topology will be ignored. Shortest path routes are also generated from any node to any other node of the network. The Gateway (next hop) field of the routes is not filled in by these configurators, so it relies on proxy ARP if the network spans several LANs. It does not perform routing table optimization (i.e. merging similar routes into a single, more general route).

Warning

The `Ipv4FlatNetworkConfigurator` is considered legacy and should not be used for new projects.

The `Ipv4FlatNetworkConfigurator` module configures the network when it is initialized. The configuration is performed in stage 2, after interface tables are filled in. Do not use an `Ipv4FlatNetworkConfigurator` module together with static routing files because they can interfere with the configurator.

The `Ipv4FlatNetworkConfigurator` searches each IP node of the network (modules that have the @node NED property and have an `Ipv4RoutingTable` submodule named "routingTable"). The configurator then assigns IP addresses to the IP nodes, controlled by the following module parameters:

- `netmask` - the common netmask of the addresses (default is 255.255.0.0).
- `networkAddress` - higher bits are the network part of the addresses, lower bits should be 0 (default is 192.168.0.0).

With the default parameters, the assigned addresses are in the range 192.168.0.1 - 192.168.255.254, so there can be a maximum of 65,534 nodes in the network. The same IP address will be assigned to each interface of the node, except the loopback interface, which always has address 127.0.0.1 (with 255.0.0.0 mask).

After assigning the IP addresses, the configurator fills in the routing tables. There are two kind of routes:

- default routes: for nodes that has only one non-loopback interface a route is added that matches with any destination address (the entry has 0.0.0.0 `host` and `netmask` fields). These are remote routes, but the gateway address is left unspecified. The delivery of the datagrams rely on the proxy ARP feature of the routers.
- direct routes following the shortest paths: for nodes that has more than one non-loopback interface a separate route is added to each IP node of the network. The outgoing interface is chosen by the shortest path to the target node. These routes are added as direct routes, even if there is no direct link with the destination. In this case proxy ARP is needed to deliver the datagrams.

Note

This configurator does not try to optimize the routing tables. If the network contains n nodes, the size of all routing tables will be proportional to n^2, and the time of the lookup of the best matching route will be proportional to n.

11.2.3 Routing Files (Legacy)

Routing files are files with `.irt` or `.mrt` extension, and their names are passed in the `routingFile` parameter to `Ipv4RoutingTable` modules.

Routing files may contain network interface configuration and static routes. Both are optional. Network interface entries in the file configure existing interfaces; static routes are added to the route table.

Warning

Routing files are considered legacy, use do not use them for new projects. Their contents can be expressed in `Ipv4NetworkConfigurator` config files.

Interfaces themselves are represented in the simulation by modules (such as the PPP module). Modules automatically register themselves with appropriate defaults in the IPv4RoutingTable, and entries in the routing file refine (overwrite) these settings. Interfaces are identified by names (e.g. ppp0, ppp1, eth0) which are normally derived from the module's name: a module called `"ppp[2]"` in the NED file registers itself as interface ppp2.

An example routing file (copied here from one of the example simulations):

```
ifconfig:

# ethernet card 0 to router
name: eth0   inet_addr: 172.0.0.3   MTU: 1500   Metric: 1  BROADCAST MULTICAST
Groups: 225.0.0.1:225.0.1.2:225.0.2.1

# Point to Point link 1 to Host 1
name: ppp0   inet_addr: 172.0.0.4   MTU: 576   Metric: 1

ifconfigend.

route:
172.0.0.2   *           255.255.255.255  H  0   ppp0
172.0.0.4   *           255.255.255.255  H  0   ppp0
default:    10.0.0.13   0.0.0.0          G  0   eth0

225.0.0.1   *           255.255.255.255  H  0   ppp0
225.0.1.2   *           255.255.255.255  H  0   ppp0
225.0.2.1   *           255.255.255.255  H  0   ppp0

225.0.0.0   10.0.0.13   255.0.0.0        G  0   eth0

routeend.
```

The `ifconfig...ifconfigend.` part configures interfaces, and `route..routeend.` part contains static routes. The format of these sections roughly corresponds to the output of the `ifconfig` and `netstat -rn` Unix commands.

An interface entry begins with a `name:` field, and lasts until the next `name:` (or until `ifconfigend.`). It may be broken into several lines.

Accepted interface fields are:

- `name:` - arbitrary interface name (e.g. eth0, ppp0)
- `inet_addr:` - IP address
- `Mask:` - netmask
- `Groups:` Multicast groups. 224.0.0.1 is added automatically, and 224.0.0.2 also if the node is a router (IPForward==true).
- `MTU:` - MTU on the link (e.g. Ethernet: 1500)
- `Metric:` - integer route metric
- flags: `BROADCAST`, `MULTICAST`, `POINTTOPOINT`

The following fields are parsed but ignored: `Bcast`, `encap`, `HWaddr`.

Interface modules set a good default for MTU, Metric (as $2*10^9$/bitrate) and flags, but leave `inet_addr` and `Mask` empty. `inet_addr` and `mask` should be set either from the routing file or by a dynamic network configuration module.

The route fields are:

```
Destination  Gateway  Netmask  Flags  Metric Interface
```

`Destination`, `Gateway` and `Netmask` have the usual meaning. The `Destination` field should either be an IP address or "default" (to designate the default route). For `Gateway`, `*` is also accepted with the meaning `0.0.0.0`.

`Flags` denotes route type:

- *H* "host": direct route (directly attached to the router), and

- *G* "gateway": remote route (reached through another router)

`Interface` is the interface name, e.g. `eth0`.

Important

The meaning of the routes where the destination is a multicast address has been changed in version 1.99.4. Earlier, these entries were used both to select the outgoing interfaces of multicast datagrams sent by the higher layer (if multicast interface was otherwise unspecified) and to select the outgoing interfaces of datagrams that are received from the network and forwarded by the node.

From version 1.99.4, multicast routing applies reverse path forwarding. This requires a separate routing table, which cannot be populated from the old routing table entries. Therefore, simulations that use multicast forwarding cannot use the old configuration files; they should be migrated to use an `Ipv4NetworkConfigurator` instead.

Some change is needed in models that use link-local multicast too. Earlier, if the IP module received a datagram from the higher layer and multiple routes were given for the multicast group, then IP sent a copy of the datagram on each interface of that route. From version 1.99.4, only the first matching interface is used (considering the longest match). If the application wants to send the multicast datagram on each interface, then it must explicitly loop and specify the multicast interface.

11.3 Configuring IPv6 Networks

IPv6 networks are configured by the `Ipv6NetworkConfigurator` module, the IPv6 counterpart of `Ipv4NetworkConfigurator`. Like it, it is a global module: a single instance assigns addresses and sets up static routes for the whole network at the start of the simulation, and takes no further part in the run.

11.3.1 Ipv6NetworkConfigurator

The `Ipv6NetworkConfigurator` assigns IPv6 prefixes and addresses, and computes and installs static routes. It discovers the network topology and its links, accepts partially specified configurations (filling in the rest automatically), and is driven by an XML document given in its `config` parameter. It shares the topology-graph model and the host-selection attributes (`@hosts`, `@names`, `@towards`, `@among`) with `Ipv4NetworkConfigurator`; see that section for the concepts common to both. As with IPv4, each node's network layer reaches the global instance through the `networkConfiguratorModule` parameter of its per-node configurator (e.g. `**.ipv6.configurator.networkConfiguratorModule = "configurator"`).

Address assignment. Two modes are available, selected by the `configureAdvPrefixes` parameter:

- *Direct assignment* (the default): every interface is given a global address derived from its link's prefix. This is the simplest setup and needs no Router Advertisements.
- *Stateless autoconfiguration* (`configureAdvPrefixes = true`): only router interfaces are configured, with the on-link prefix they advertise; hosts then form their own global addresses by SLAAC from the Router Advertisements (see *The IPv6 Protocol Family*). This more closely models a real IPv6 network.

The `<interface>` element selects interfaces and sets their `prefix` (e.g. `prefix='2001:db8:1::/64'`; an `x` group lets the configurator fill in a unique value), `mtu`, and – on routers – the Router Advertisement flags and lifetimes (`advValidLifetime`, `advPreferredLifetime`, `advOnLinkFlag`, `advAutonomousFlag`, `advManagedFlag`, `advOtherConfigFlag`, `advDefaultLifetime`), the RA timing (`maxRtrAdvInterval`, `minRtrAdvInterval`) and `dupAddrDetectTransmits`. For example, Mobile IPv6 needs fast RAs on wireless-facing router interfaces:

```
<config>
  <interface among='Home_Agent R_1' names='eth1'
             maxRtrAdvInterval='0.07' minRtrAdvInterval='0.03'/>
</config>
```

Routes. When addStaticRoutes is set, the configurator computes shortest-path routes between all nodes, and adds default and on-link routes where appropriate. Routes may also be given explicitly with <route> elements (attributes hosts, destination, gateway, interface, metric). The destination prefix length may be supplied either as a separate prefixLength attribute or appended to destination in CIDR notation:

```
<config>
  <interface among='*' prefix='2001:db8:1::/64'/>
  <route hosts='h1' destination='2001:db8:2::/64' interface='eth0'/>
</config>
```

An <autoroute> element tunes the automatic route generation (source and destination selection, link metrics), analogously to Ipv4NetworkConfigurator.

Parameters. The most commonly used parameters are:

- config: the XML configuration document (<interface>, <route> and <autoroute> elements).
- assignAddresses: assign global addresses to interfaces (default true).
- configureAdvPrefixes: use SLAAC mode – configure router AdvPrefixes only and let hosts autoconfigure (default false).
- addStaticRoutes, addDefaultRoutes, addDirectRoutes: enable the kinds of automatic route generation (all default true).
- dumpAddresses, dumpRoutes, dumpTopology, dumpLinks: print the assigned addresses, the routing tables, and the discovered topology and links, for debugging (all default false).

11.4 Configuring Layer 2

The L2NetworkConfigurator module allows configuring network scenarios at layer 2. The STP/RTP-related parameters such as link cost, port priority and the "is-edge" flag can be configured with XML files.

This module is similar to Ipv4NetworkConfigurator. It supports the selector attributes @hosts, @names, @towards, @among, and they behave similarly to its Ipv4NetworkConfigurator equivalent. The @ports selector is also supported, for configuring per-port parameters.

The following example configures port 5 (if it exists) on all switches, and sets cost=19 and priority=32768:

```
<config>
  <interface hosts='**' ports='5' cost='19' priority='32768'/>
</config>
```

For more information about the usage of the selector attributes, see Ipv4NetworkConfigurator.

11.5 Configuring Time-Sensitive Networking

Time-Sensitive Networking (TSN) requires additional network-wide configuration on top of the addressing and routing covered above: periodic gate schedules for the time-aware shapers, redundant streams, and the protection of streams against failures. INET provides a family of TSN configurators for this; like the configurators above, each discovers the network topology and then computes and installs the parameters of the underlying per-node modules.

- *Gate scheduling* configurators compute the gate control lists (the periodic open/close states of the per-class transmission gates) from the stream packet lengths, intervals and latency requirements. Several implementations sharing GateScheduleConfiguratorBase are available: EagerGateScheduleConfigurator, Z3GateScheduleConfigurator (using a SAT solver), and TSNschedGateScheduleConfigurator (using the external TSNsched tool).
- StreamRedundancyConfigurator sets up stream identification, splitting and merging across the network to form the redundant streams used by frame replication and elimination (IEEE 802.1CB).
- FailureProtectionConfigurator drives the gate-scheduling and stream-redundancy configurators together to protect every stream against a specified set of link and node failures.

- `MacForwardingTableConfigurator` fills in static MAC forwarding tables (an alternative to MAC address learning) for a bridged network.

These work alongside the IPv4/IPv6 and `L2NetworkConfigurator` configurators described above. Their XML configuration and use are covered in detail in the *Time-Sensitive Networking* chapter.

CHAPTER

TWELVE

INTERNET ROUTING

12.1 Overview

The INET Framework has models for several internet routing protocols, including RIP, OSPF, and BGP.

The easiest way to add routing to a network is to use the `Router` NED type for routers. The `Router` type contains a conditional instance for each of the above protocols. These submodules can be enabled by setting the `hasRip`, `hasOspf`, and/or `hasBgp` parameters to `true`.

Example:

```
**.hasRip = true
```

There are also NED types called `RipRouter`, `OspfRouter`, and `BgpRouter`, which are all variants of the `Router` module with the respective routing protocols enabled.

12.2 RIP

RIP (Routing Information Protocol) is a distance-vector routing protocol that employs the hop count as a routing metric. Routing loops are prevented in RIP by implementing a hop limit in the path from the source to the destination. RIP uses the *split horizon with poison reverse* technique to address the "count-to-infinity" problem.

The `Rip` module implements distance vector routing as specified in RFC 2453 (RIPv2) and RFC 2080 (RIPng). The behavior can be selected by setting the `mode` parameter to either *RIPv2* ` or `RIPng`. Protocol configuration such as link metrics and per-interface operation mode (such as whether RIP is enabled on the interface and whether to use split horizon) can be specified in XML using the `ripConfig` parameter.

The following example configures a `Router` module to use RIPv2:

```
**.hasRip = true
**.mode = "RIPv2"
**.ripConfig = xmldoc("RIPConfig.xml")
```

The configuration file specifies the per-interface parameters. Each <`interface`> element configures one or more interfaces; the `hosts`, `names`, `towards`, `among` attributes select the configured interfaces (similar to `Ipv4NetworkConfigurator`). See the *Network Autoconfiguration* chapter for further information.

Additional attributes:

- `metric`: metric assigned to the link, with a default value of 1. This value is added to the metric of a learned route received on this interface. It must be an integer in the [1,15] interval.
- `mode`: mode of the interface.

The mode attribute can be one of the following:

- `NoRIP`: no RIP messages are sent or received on this interface.
- `NoSplitHorizon`: no split horizon filtering; send all routes to neighbors.
- `SplitHorizon`: do not send routes whose next hop is the neighbor.

- `SplitHorizonPoisonedReverse` (default): if the next hop is the neighbor, then set the metric of the route to infinity.

The following example sets the link metric between router `R1` and `RB` to 2, while all other links will have a metric of 1.

```
<RIPConfig>
  <interface among="R1 RB" metric="2"/>
  <interface among="R? R?" metric="1"/>
</RIPConfig>
```

12.3 OSPF

OSPF (Open Shortest Path First) is a routing protocol for IP networks. It uses a link state routing (LSR) algorithm and falls into the group of interior gateway protocols (IGPs), operating within a single autonomous system (AS).

`OspfRouter` is a `Router` with the OSPF protocol enabled.

The `Ospfv2` module implements OSPF protocol version 2. Areas and routers can be configured using an XML file specified by the `ospfConfig` parameter. Various parameters for the network interfaces can also be specified in the XML file or as a parameter of the `Ospfv2` module.

```
**.ospf.ospfConfig = xmldoc("ASConfig.xml")
**.ospf.helloInterval = 12s
**.ospf.retransmissionInterval = 6s
```

The <OSPFASConfig> root element may contain <Area> and <Router> elements with various attributes specifying the parameters for the network interfaces. Most importantly, <Area> contains <AddressRange> elements enumerating the network addresses that should be advertised by the protocol. Also, <Router> elements may contain data for configuring various point-to-point or broadcast interfaces.

```
<?xml version="1.0"?>
<OSPFASConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"␣
↪xsi:schemaLocation="OSPF.xsd">
  <!-- Areas -->
  <Area id="0.0.0.0">
    <AddressRange address="H1" mask="H1" status="Advertise" />
    <AddressRange address="H2" mask="H2" status="Advertise" />
    <AddressRange address="R1>R2" mask="R1>R2" status="Advertise" />
    <AddressRange address="R2>R1" mask="R2>R1" status="Advertise" />
  </Area>
  <!-- Routers -->
  <Router name="R1" RFC1583Compatible="true">
    <BroadcastInterface ifName="eth0" areaID="0.0.0.0" interfaceOutputCost="1"␣
↪routerPriority="1" />
    <PointToPointInterface ifName="eth1" areaID="0.0.0.0" interfaceOutputCost="2" />
  </Router>
  <Router name="R2" RFC1583Compatible="true">
    <PointToPointInterface ifName="eth0" areaID="0.0.0.0" interfaceOutputCost="2" />
    <BroadcastInterface ifName="eth1" areaID="0.0.0.0" interfaceOutputCost="1"␣
↪routerPriority="2" />
  </Router>
</OSPFASConfig>
```

12.4 BGP

BGP (Border Gateway Protocol) is a standardized exterior gateway protocol designed to exchange routing and reachability information among autonomous systems (AS) on the Internet.

`BgpRouter` is a `Router` with the BGP protocol enabled.

The `Bgp` module implements BGP Version 4. The model implements RFC 4271, with some limitations. Autonomous Systems and rules can be configured in an XML file that can be specified in the `bgpConfig` parameter.

```
**.bgpConfig = xmldoc("BGPConfig.xml")
```

The configuration file may contain <AS> and <Session> elements at the top level.

- <AS>: defines Autonomous Systems, routers, and rules to be applied.
- <Session>: specifies open sessions between edge routers. It must contain exactly two <Router exterAddr="x.x.x.x"/> elements.

The BGP timers are module parameters of the `Bgp` module, not part of the XML: `connectRetryTime` (default 120s), `holdTime` (180s), `keepAliveTime` (60s) and `startDelay` (the base session-startup delay). For example:

```
**.bgp.holdTime = 90s
```

(Earlier versions configured these in a <TimerParams> XML element, which is no longer accepted.)

```
<BGPConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="BGP.xsd">

  <AS id="60111">
    <Router interAddr="172.1.10.255"/> <!--Router A1-->
    <Router interAddr="172.1.20.255"/> <!--Router A2-->
  </AS>

  <AS id="60222">
    <Router interAddr="172.10.4.255"/> <!--Router B-->
  </AS>

  <AS id="60333">
    <Router interAddr="172.13.1.255"/> <!--Router C1-->
    <Router interAddr="172.13.2.255"/> <!--Router C2-->
    <Router interAddr="172.13.3.255"/> <!--Router C3-->
    <Router interAddr="172.13.4.255"/> <!--Router C4-->
    <DenyRouteOUT Address="172.10.8.0" Netmask="255.255.255.0"/>
    <DenyASOUT> 60111 </DenyASOUT>
  </AS>

  <Session id="1">
    <Router exterAddr="10.10.10.1" > </Router> <!--Router A1-->
    <Router exterAddr="10.10.10.2" > </Router> <!--Router C1-->
  </Session>

  <Session id="2">
    <Router exterAddr="10.10.20.1" > </Router> <!--Router A2-->
    <Router exterAddr="10.10.20.2" > </Router> <!--Router B-->
  </Session>

  <Session id="3">
    <Router exterAddr="10.10.30.1" > </Router> <!--Router B-->
    <Router exterAddr="10.10.30.2" > </Router> <!--Router C2-->
```

(continues on next page)

(continued from previous page)

```
  </Session>
</BGPConfig>
```

Inside <AS> elements, various rules can be specified:

- DenyRoute: deny route in IN and OUT traffic (`Address` and `Netmask` attributes must be specified.)
- DenyRouteIN: deny route in IN traffic (`Address` and `Netmask` attributes must be specified.)
- DenyRouteOUT: deny route in OUT traffic (`Address` and `Netmask` attributes must be specified.)
- DenyAS: deny routes learned by AS in IN and OUT traffic (AS id must be specified as the body of the element.)
- DenyASIN: deny routes learned by AS in IN traffic (AS id must be specified as the body of the element.)
- DenyASOUT: deny routes learned by AS in OUT traffic (AS id must be specified as the body of the element.)

12.4.1 BGP over IPv6 (MP-BGP)

BGP can also run over IPv6 and carry IPv6 routes, using the Multiprotocol Extensions of RFC 4760. To configure an IPv6 BGP speaker, set the `addressFamily` parameter of the `Bgp` module to `"ipv6"`, point its `routingTableModule` at the node's IPv6 routing table, and set the `routerId` parameter – the 4-octet BGP Identifier (RFC 4271/6286), which must be configured explicitly because an IPv6-only router has no IPv4 address to derive it from. The addresses in the BGP configuration file are then IPv6 addresses, and IPv6 reachability is exchanged in `MP_REACH_NLRI` / `MP_UNREACH_NLRI` path attributes. A single `Bgp` instance serves one address family.

```
*.routerX.hasBgp = true
*.routerX.hasIpv6 = true
*.routerX.bgp.addressFamily = "ipv6"
*.routerX.bgp.routingTableModule = "^.ipv6.routingTable"
*.routerX.bgp.routerId = "1.1.1.1"
```

CHAPTER

THIRTEEN

AD HOC ROUTING

13.1 Overview

In ad hoc networks, the topology of the network is not known to the nodes. Instead, the topology must be discovered. When a new node joins the network, it announces its presence and listens for announcements from its neighbors. Each node learns about nearby nodes and how to reach them, and it may also announce that it can reach other nodes. The task of routing in ad hoc networks is complicated by the fact that nodes may be mobile, resulting in a changing topology.

There are two types of ad hoc routing protocols: proactive and reactive. *Proactive* or *table-driven* protocols maintain up-to-date lists of destinations and their routes by periodically distributing routing tables throughout the network. *Reactive* or *on-demand* protocols find a route on-demand by flooding the network with Route Request packets.

The INET Framework includes the implementation of several ad hoc routing protocols, including AODV, DSDV, DYMO, and GPSR.

The easiest way to add routing to an ad hoc network is to use the `ManetRouter` NED type for nodes. `ManetRouter` contains a submodule named `routing` which has parametric type, so it can be configured to be an AODV router, a DYMO router, or a router for any other supported routing protocol. For example, `ManetRouter` nodes in the network can be configured to use AODV by adding the following line to the ini file:

```
**.routingApp.typename = "Aodv" # as an application
**.routing.typename = "Gpsr" # as a routing protocol module
```

There are also NED types called `AodvRouter`, `DymoRouter`, `DsdvRouter`, and `GpsrRouter`. These types are all based on `ManetRouter` and have the routing protocol submodule type set appropriately.

13.2 AODV

AODV (Ad hoc On-Demand Distance Vector Routing) is a routing protocol for mobile ad hoc networks and other wireless ad hoc networks. It provides quick adaptation to dynamic link conditions, low processing and memory overhead, low network utilization, and determines unicast routes to destinations within the ad hoc network.

The `Aodv` module type implements AODV, based on RFC 3561.

`AodvRouter` is a `ManetRouter` with the routing module type set to `Aodv`.

13.3 DSDV

DSDV (Destination-Sequenced Distance-Vector Routing) is a table-driven routing scheme for ad hoc mobile networks based on the Bellman-Ford algorithm.

The `Dsdv` module type implements DSDV. It is currently a partial implementation.

`DsdvRouter` is a `ManetRouter` with the routing module type set to `Dsdv`.

13.4 DYMO

The DYMO (Dynamic MANET On-demand) routing protocol is the successor to the AODV routing protocol. DYMO can work as both a proactive and a reactive routing protocol, i.e. routes can be discovered just when they are needed.

The `Dymo` module type implements DYMO, based on the IETF draft "draft-ietf-manet-dymo-24".

`DymoRouter` is a `ManetRouter` with the routing module type set to `Dymo`.

13.5 GPSR

GPSR (Greedy Perimeter Stateless Routing) is a routing protocol for mobile wireless networks that uses the geographic positions of nodes to make packet forwarding decisions.

The `Gpsr` module type implements GPSR, based on the paper "GPSR: Greedy Perimeter Stateless Routing for Wireless Networks" by Brad Karp and H. T. Kung, 2000. The implementation supports both GG and RNG planarization algorithms.

`GpsrRouter` is a `ManetRouter` with the routing module type set to `Gpsr`.

CHAPTER

FOURTEEN

QUEUEING MODEL

14.1 Overview

The INET queueing model provides reusable modules for various application areas. These modules can be used to build application traffic generators, queueing models for MAC protocols, traffic conditioning models for quality of service implementations, and so on.

14.1.1 Usage

The queueing modules can be used in two very different ways. Firstly, they can be connected to other INET modules using gates. In this case, the modules send and receive packets asynchronously, as many other INET modules do. For example, application packet source and packet sink modules are used this way. The other way to use them is to directly call their C++ methods through one of the C++ interfaces of the contract package. In this case, the queueing modules are not connected to other INET modules at all. For example, MAC protocol modules use packet queues as submodules through C++ method calls.

14.1.2 Model

Most queueing model elements provide simple behaviors, so they are implemented as simple modules. However, queueing elements can also be composed to form more complex behaviors. For example, priority queues, request-response based traffic generators, traffic shapers are usually implemented as compound modules. In fact, some of the queueing model elements provided by INET are actually realized as compound modules using composition.

The queueing model can be found in the `inet.queueing` NED package. All queueing model elements implement one or more NED module interfaces and also the corresponding C++ interfaces from the contract folder. As a minimum, they all implement the `IPacketQueueingElement` C++ interface.

14.1.3 Operation

Internally, connected queueing model elements most often communicate with each other using synchronous C++ method calls, without utilizing `handleMessage()`. The only exception is when an operation takes a non-zero simulation time, such as when the processing of a packet is delayed. The connections between the model elements are simply used to designate the caller and the callee. Other modules can still send and receive messages through the gates of queueing model elements, but only `Packet` instances are allowed.

There are two new important operations on queueing model elements defined in `IPassivePacketSource` and `IPassivePacketSink`. Packets can be *pushed* into gates and packets can be *pulled* from gates. Both of these operations are synchronous, they either finish successfully or throw an exception. The main difference between pushing and pulling is the subject of the activity. In the former case, when a packet is pushed, the activity is initiated by the source of the packet. In contrast, when a packet is pulled, the activity is initiated by the sink of the packet. In both cases, the passive elements can be asked via separate methods to tell if they can be pushed or pulled on a specific gate in their current state, either with respect to a specific packet or any packet thereof.

Queueing model elements can be divided into two categories with respect to the operation on a given gate: *passive* and *active*. The passive model elements are pushed into and pulled from by other connected modules. In contrast, the active model elements push into and pull from other connected modules as they see fit. They implement the interfaces `IActivePacketSource` and `IActivePacketSink`.

The active queueing elements take into consideration the state of the connected passive elements. That is, they push or pull packets only when the passive end is able to consume or provide, respectively. The queueing model elements also validate the assembled structure during module initialization with respect to the active and passive behavior of the connected gates. That is, active output gates must be connected to passive input gates, and active input gates must be connected to passive output gates.

The mechanism that allows passive modules to prevent a push or pull operation of an active module is called backpressure. For example, a queue that is full, and which doesn't have a configured packet dropping mechanism generates backpressure towards the connected active packet source. This backpressure prevents the source from pushing more packets into the queue. Similarly, an empty queue generates backpressure towards the connected active server component. The backpressure prevents the server from pulling more packets from the queue. In general, backpressure can be generated by modules on their own at any time they want; it can be transferred from one gate to another; or it can be simply absorbed by a module. For example, a scheduler transfers the backpressure from its input gates towards its output gate, and similarly, a classifier transfers backpressure from its output gates towards its input gate. Finally, modules can choose if they consider or ignore the backpressure from the connected modules. In the latter case, the push and pull operations may cause a runtime error.

Pushing a packet into a gate (and similarly pulling a packet from a gate) can have far-reaching consequences by triggering a chain of operations potentially by passing through multiple connected elements. For example, pushing a packet into a classifier will immediately push the packet into one of the connected queueing elements, and will also potentially lead to additional operations on further connected elements. Similarly, pulling a packet from a scheduler will immediately pull a packet from one of the connected queueing elements, and will also potentially lead to additional operations on further connected elements.

In general, the following equation about the number of packets holds true for all queueing model elements:

#created + #pushed - #pulled - #removed - #dropped = #available + #delayed

14.2 Sources

These modules act as sources of packets. An active packet source pushes packets to its output. A passive packet source returns a packet when it is pulled by other queueing model elements.

- `ActivePacketSource`: a generic source that produces packets periodically
- `PassivePacketSource`: a generic source that provides packets as requested
- `BurstyPacketProducer`: mixes two different sources to generate bursty traffic
- `QueueFiller`: produces packets to continuously fill a queue
- `ResponseProducer`: produces complex response traffic based on the incoming request type
- `PcapFilePacketProducer`: replays packets from a PCAP file
- `EmptyPacketSource`: doesn't push packets and doesn't allow packets to be pulled

14.3 Sinks

These modules act as sinks of packets. An active packet sink pulls packets from its input. A passive packet sink is pushed with packets by other queueing model elements.

- `ActivePacketSink`: a generic sink that collects packets periodically
- `PassivePacketSink`: a generic sink that consumes packets as they arrive
- `RequestConsumer`: processes incoming requests in order and initiates response traffic
- `PcapFilePacketConsumer`: writes packets to a PCAP file
- `FullPacketSink`: doesn't pull packets and doesn't allow packets to be pushed

14.4 Queues

These modules store packets and maintain an ordering among them. Queues do not delay packets, so if a queue is not empty, then a packet is always available. When a packet is pushed into the input of a queue, then the packet is either stored, or if the queue is overloaded, it is dropped. When a packet is pulled from the output of a queue, then one of the stored packets is returned.

The following simpler equation about the number of packets always holds true for queues:

#pushed - #pulled - #dropped - #removed = #queueLength = #available

- `PacketQueue`: generic queue that provides ordering and selective dropping; it can be parameterized with a *comparator* and a *dropper* function (such functions can be defined in C++)
- `DropHeadQueue`: drops packets at the head of the queue
- `DropTailQueue`: drops packets at the tail of the queue, the most commonly used queue
- `PriorityQueue`: contains several subqueues that share a buffer
- `RedDropperQueue`: combines random early detection with a queue
- `CompoundPacketQueueBase`: allows building complex queues by pure NED composition

14.5 Buffers

These modules deal with memory allocation of packets without considering the ordering among them. A packet buffer generally doesn't have gates, and packets are not pushed into or pulled from it.

- `PacketBuffer`: generic buffer that provides shared storage between several queues; it can be parameterized with a *dropper* function (such functions can be defined in C++)
- `PriorityBuffer`: drops packets based on the queue priority

14.6 Filters

These modules filter specific packets while dropping the rest. When a packet is pushed into the input of a packet filter, then the filter either pushes the packet to its output or it simply drops the packet. In contrast, when a packet is pulled from the output of a packet filter, then it continuously pulls and drops packets from its input until it finds one that matches the filter criteria.

- `PacketFilter`: generic packet filter; it can be parameterized with a *filter* function (such functions can be defined in C++)
- `ContentBasedFilter`: drops packets based on the data they contain
- `OrdinalBasedDropper`: drops packets based on their ordinal number
- `StatisticalRateLimiter`: drops packets above the specified packetrate or datarate
- `RedDropper`: drops packets based on random early detection

14.7 Classifiers

These modules classify packets to one of their outputs. When a packet is pushed into the input of a packet classifier, then it immediately pushes the packet to one of its outputs.

- `PacketClassifier`: generic packet classifier; it can be parameterized with a *classifier* function (such functions can be defined in C++)
- `PriorityClassifier`: classifies packets to the first non-full output
- `WrrClassifier`: classifies packets in a weighted round-robin manner
- `LabelClassifier`: classifies packets based on the attached labels

- `MarkovClassifier`: classifies packets based on the state of a Markov process
- `UserPriorityClassifier`: classifies packets based on the attached `UserPriorityReq`
- `ContentBasedClassifier`: classifies packets based on the data they contain

14.8 Schedulers

These modules schedule packets from one of their inputs. When a packet is pulled from the output of a packet scheduler, then it immediately pulls a packet from one of its inputs and returns that packet.

- `PacketScheduler`: generic packet scheduler; it can be parameterized with a *packet scheduler* function (such functions can be defined in C++)
- `PriorityScheduler`: schedules packets from the first non-empty source
- `WrrScheduler`: schedules packets in a weighted round-robin manner
- `LabelScheduler`: schedules packets based on the attached labels
- `MarkovScheduler`: schedules packets based on the state of a Markov process
- `ContentBasedScheduler`: schedules packets based on the data they contain

14.9 Servers

These modules process packets in order one by one. A packet server actively pulls packets from its input when it sees fit, and it also actively pushes packets into its output.

- `PacketServer`: serves packets according to the processing time based on packet length
- `TokenBasedServer`: serves packets when the required number of tokens are available (token generators are described later)

14.10 Markers

These modules attach some information to packets on an individual basis. Packets can be both pushed into the input and pulled from the output of packet markers.

- `PacketLabeler`: generic marker which attaches labels to matching packets; it can be parameterized with an `IPacketFilterFunction`
- `ContentBasedLabeler`: attaches labels to packets based on the data they contain
- `PacketTagger`: attaches tags such as outgoing interface, hopLimit, VLAN, user priority to matching packets; it can be parameterized with an `IPacketFilterFunction`
- `ContentBasedTagger`: attaches tags to packets based on the data they contain

14.11 Meters

These modules measure some property of a stream of packets. Packets can be both pushed into the input and pulled from the output of packet meters.

- `ExponentialRateMeter`: Measures the packetrate and datarate of the packet stream. Takes older packets into account with an exponentially decreasing weight
- `SlidingWindowRateMeter`: Measures the packetrate and datarate of the packet stream. Takes packets into account only in the given time window

14.12 Token generators

These modules generate tokens for other modules. A token generator generally doesn't have gates, and packets are not pushed into or pulled from it.

- `TimeBasedTokenGenerator`: generates tokens based on elapsed simulation time
- `PacketBasedTokenGenerator`: generates tokens based on received packets
- `SignalBasedTokenGenerator`: generates tokens based on received signals
- `QueueBasedTokenGenerator`: generates tokens based on the state of a queue

14.13 Conditioners

These modules actively shape traffic by changing the order of packets, dropping packets, delaying packets, etc. Note that the capabilities of conditioners also include delaying, which queues are not capable of. Traffic conditioners are generally built by composition using other queueing model elements.

- `LeakyBucket`: a generic shaper with overflow and configurable output rate
- `TokenBucket`: a generic shaper with overflow and configurable burstiness and output rate

14.14 Other queueing elements

There are also some other generic queueing model elements which don't fit well into any of the above categories.

- `PacketGate`: allows or prevents packets to pass through, either pushed or pulled
- `PacketMultiplexer`: passively connects multiple inputs to a single output, packets are pushed into the inputs
- `PacketDemultiplexer`: passively connects a single input to multiple outputs, packets are pulled from the outputs
- `PacketDelayer`: sends received packets to the output with some delay independently
- `PacketCloner`: sends one copy of each received packet to all outputs
- `PacketHistory`: keeps track of the last N packets which can be inspected in Qtenv
- `PacketDuplicator`: sends copies of each received packet to the only output
- `OrdinalBasedDuplicator`: duplicates received packets based on their ordinal number

CHAPTER

FIFTEEN

DIFFERENTIATED SERVICES

15.1 Overview

In the early days of the Internet, only best effort service was defined. The Internet delivers each packet individually, and delivery time is not guaranteed. Moreover, packets may even be dropped due to congestion at the routers of the network. It was assumed that transport protocols and applications can overcome these deficiencies. This worked until FTP and email were the main applications of the Internet, but the newer applications such as Internet telephony and video conferencing cannot tolerate delay jitter and loss of data.

The first attempt to add QoS capabilities to IP routing was Integrated Services. Integrated services provide resource assurance through resource reservation for individual application flows. An application flow is identified by the source and destination addresses and ports and the protocol ID. Before data packets are sent, the necessary resources must be allocated along the path from the source to the destination. Each router along the path must examine the packets and decide if they belong to a reserved application flow. This could cause a memory and processing demand on the routers. Another drawback is that the reservation must be periodically refreshed, so there is overhead during the data transmission as well.

Differentiated Services is a more scalable approach that offers a better than best-effort service. Differentiated Services do not require resource reservation setup. Instead of making per-flow reservations, Differentiated Services divides the traffic into a small number of *forwarding classes*. The forwarding class is directly encoded into the packet header. After packets are marked with their forwarding classes at the edge of the network, the interior nodes of the network can use this information to differentiate the treatment of packets. The forwarding classes may indicate drop priority and resource priority. For example, when a link is congested, the network will drop packets with the highest drop priority first.

In the Differentiated Service architecture, the network is partitioned into DiffServ domains. Within each domain, the resources are allocated to forwarding classes, taking into account the available resources and the traffic flows. There are *service level agreements* (SLAs) between the users and service providers, and between the domains that describe the mapping of packets to forwarding classes and the allowed traffic profile for each class. The routers at the edge of the network are responsible for marking the packets and protecting the domain from misbehaving traffic sources. Nonconforming traffic may be dropped, delayed, or marked with a different forwarding class.

15.1.1 Implemented Standards

The implementation follows these RFCs:

- RFC 2474: Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers
- RFC 2475: An Architecture for Differentiated Services
- RFC 2597: Assured Forwarding PHB Group
- RFC 2697: A Single Rate Three Color Marker
- RFC 2698: A Two Rate Three Color Marker
- RFC 3246: An Expedited Forwarding PHB (Per-Hop Behavior)
- RFC 3290: An Informal Management Model for Diffserv Routers

15.2 Architecture of NICs

Network interface modules, such as `PppInterface` and `EthernetInterface`, may contain traffic conditioners in their input and output data path.

Network interfaces may also contain an optional external queue component. In the absence of an external queue module, `Ppp` and `EthernetCsmaMacPhy` use an internal drop-tail queue to buffer the packets while the line is busy.

15.2.1 Traffic Conditioners

Traffic conditioners have one input and one output gate as defined in the `ITrafficConditioner` interface. They can transform the incoming traffic by dropping or delaying packets. They can also set the DSCP field of the packet or mark it in another way for differentiated handling in the queues.

Traffic conditioners perform the following actions:

- Classify the incoming packets.
- Meter the traffic in each class.
- Mark or drop packets depending on the result of metering.
- Shape the traffic by delaying packets to conform to the desired traffic profile.

INET provides classifier, meter, and marker modules that can be composed to build a traffic conditioner as a compound module.

15.2.2 Output Queues

Queue components must implement the `IPacketQueue` module interface. In addition to having one input and one output gate, these components must implement a passive queue behavior: they only deliver a packet when the module connected to their output explicitly requests it. (In C++ terms, the module must implement the `IPacketQueue` interface. The next module requests a packet by calling the `requestPacket()` method of that interface.)

15.3 Simple modules

This section describes the primitive elements from which traffic conditioners and output queues can be built. The following sections show some examples of how these queues, schedulers, droppers, classifiers, meters, and markers can be combined.

The types of the components are:

- `queue`: container of packets, accessed as FIFO.
- `dropper`: attached to one or more queues, it can limit the queue length below some threshold by selectively dropping packets.
- `scheduler`: decides which packet is transmitted first when more packets are available on their inputs.
- `classifier`: classifies the received packets according to their content (e.g., source/destination address and port, protocol, DSCP field of IP datagrams) and forwards them to the corresponding output gate.
- `meter`: classifies the received packets according to the temporal characteristic of their traffic stream.
- `marker`: marks packets by setting their fields to control their further processing.

15.3.1 Queues

When packets arrive at a higher rate than the interface can transmit, they are queued.

Queue elements store packets until they can be transmitted. They have one input and one output gate. Queues may have one or more thresholds associated with them.

Received packets are enqueued and stored until the module connected to their output requests a packet by calling the `requestPacket()` method.

They should be able to notify the module connected to their output about the arrival of new packets.

FIFO Queue

The `PacketQueue` module implements a passive FIFO queue with unlimited buffer space. It can be combined with algorithmic droppers and schedulers to form an IPacketQueue compound module.

The C++ class implements the `IQueueAccess` and `IPacketQueue` interfaces.

DropTailQueue

The other primitive queue module is `DropTailQueue`. Its capacity can be specified by the `packetCapacity` parameter. When the number of stored packets reaches the capacity of the queue, further packets are dropped. Because this module contains a built-in dropping strategy, it cannot be combined with algorithmic droppers as `PacketQueue` can be. However, its output can be connected to schedulers.

This module implements the `IPacketQueue` interface, so it can be used as the queue component of an interface card.

15.3.2 Droppers

Algorithmic droppers selectively drop received packets based on some condition. The condition can be either deterministic (e.g., to bound the queue length) or probabilistic (e.g., RED queues).

Another kind of dropper is an absolute dropper that drops each received packet. It can be used to discard excess traffic, i.e. packets whose arrival rate exceeds the allowed maximum. In INET, the `Sink` module can be used as an absolute dropper.

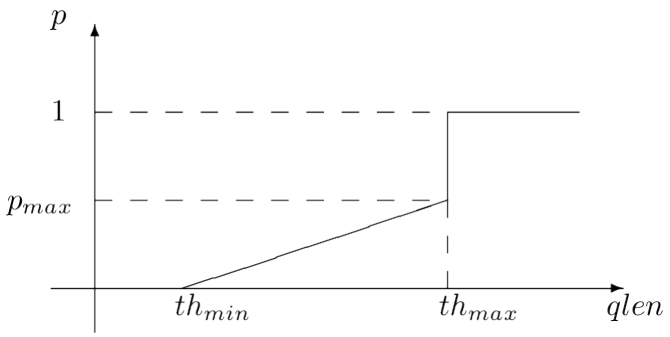
The algorithmic droppers in INET are `OrdinalBasedDropper` and `RedDropper`. These modules have multiple input and multiple output gates. Packets that arrive on gate `in[i]` are forwarded to gate `out[i]` (unless they are dropped). However, the queues attached to the output gates are viewed as a whole, i.e. the queue length parameter of the dropping algorithm is the sum of the individual queue lengths. This way, we can emulate shared buffers of the queues. Note that it is also possible to connect each output to the same queue module.

RED Dropper

The `RedDropper` module implements Random Early Detection ([**?**]).

It has n input and n output gates (specified by the `numGates` parameter). Packets that arrive at the i^{th} input gate are forwarded to the i^{th} output gate, or dropped. The output gates must be connected to simple modules implementing the `IPacketQueue` C++ interface (e.g. `PacketQueue`).

The module sums the used buffer space of the queues attached to the output gates. If it is below a minimum threshold, the packet won't be dropped, if above a maximum threshold, it will be dropped, if it is between the minimum and maximum threshold, it will be dropped by a given probability. This probability determined by a linear function which is 0 at the minth and maxp at maxth.



The queue length can be smoothed by specifying the `wq` parameter. The average queue length used in the tests are computed by the formula:

$$avg = (1 - wq) * avg + wq * qlen$$

The minth, maxth, and maxp parameters can be specified separately for each input gate, so this module can be used to implement different packet drop priorities.

15.3.3 Schedulers

Scheduler modules decide which queue can send a packet when the interface is ready to transmit one. They have several input gates and one output gate.

Modules that are connected to the inputs of a scheduler must implement the IPacketQueue C++ interface. Schedulers also implement the IPacketQueue interface, so they can be cascaded to other schedulers and used as the output module of IPacketQueue's.

There are several possible scheduling disciplines (first come/first served, priority, weighted fair, weighted round-robin, deadline-based, rate-based). INET contains an implementation of priority and weighted round-robin schedulers.

Priority Scheduler

The PriorityScheduler module implements a strict priority scheduler. Packets that arrive on in[0] have the highest priority, then packets that arrive on in[1], and so on. If multiple packets are available when one is requested, then the one with the highest priority is chosen. Packets with lower priority are transmitted only when there are no packets on the inputs with higher priorities.

PriorityScheduler must be used with care because a large volume of higher priority packets can starve lower priority packets. Therefore, it is necessary to limit the rate of higher priority packets to a fraction of the output data rate.

PriorityScheduler can be used to implement the EF PHB.

Weighted Round Robin Scheduler

The WrrScheduler module implements a weighted round-robin scheduler. The scheduler visits the input gates in turn and selects the number of packets for transmission based on their weight.

For example, if the module has three input gates and the weights are 3, 2, and 1, then packets are transmitted in this order:

```
A, A, A, B, B, C, A, A, A, B, B, C, ...
```

where A denotes packets that arrived on in[0], B denotes packets that arrived on in[1], and C denotes packets that arrived on in[2]. If there are no packets in the current one when a packet is requested, then the next one is chosen if it has enough tokens.

If the sizes of the packets are equal, then WrrScheduler divides the available bandwidth according to the weights. In each case, it allocates the bandwidth fairly. Each flow receives a guaranteed minimum bandwidth, which is ensured even if other flows exceed their share (flow isolation). It also efficiently uses the channel because if some traffic is smaller than its share of bandwidth, then the rest is allocated to the other flows.

WrrScheduler can be used to implement the AFxy PHBs.

15.3.4 Classifiers

Classifier modules have one input gate and multiple output gates. They examine received packets and forward them to the appropriate output gate based on the contents of specific packet headers. More information about classifiers can be found in RFC 2475 and RFC 3290.

The inet.networklayer.diffserv package contains two classifiers: MultiFieldClassifier to classify the packets at the edge routers of the DiffServ domain, and BehaviorAggregateClassifier to classify the packets at the core routers.

Multi-field Classifier

The `MultiFieldClassifier` module can be used to identify micro-flows in the incoming traffic. The flow is identified by the source and destination addresses, the protocol ID, and the source and destination ports of the IP packet.

The classifier can be configured by specifying a list of filters. Each filter can specify a source/destination address mask, a protocol, a source/destination port range, and bits of the TypeOfService/TrafficClass field to be matched. They also specify the index of the output gate to which matching packets should be forwarded. The first matching filter determines the output gate; if there are no matching filters, then `defaultOut` is chosen.

The configuration of the module is given as an XML document. The document element must contain a list of <filter> elements. Each filter must have a mandatory `@gate` attribute specifying the index of the output gate for packets matching the filter. Other attributes are optional and specify the conditions for a match:

- `@srcAddress`, `@srcPrefixLength`: to match the source address of the IP.
- `@destAddress`, `@destPrefixLength`:
- `@protocol`: matches the protocol field of the IP packet. Its value can be a name (e.g., "udp", "tcp") or the numeric code of the protocol.
- `@tos`, @tosMask: matches bits of the TypeOfService/TrafficClass field of the IP packet.
- `@srcPort`: matches the source port of the TCP or UDP packet.
- `@srcPortMin`, `@srcPortMax`: matches a range of source ports.
- `@destPort`: matches the destination port of the TCP or UDP packet.
- `@destPortMin`, `@destPortMax`: matches a range of destination ports.

The following example configuration specifies

- to transmit packets received from the 192.168.1.x subnet on gate 0,
- to transmit packets addressed to port 5060 on gate 1,
- to transmit packets having CS7 in their DSCP field on gate 2,
- and to transmit other packets on `defaultGate`.

```
<filters>
  <filter srcAddress="192.168.1.0" srcPrefixLength="24" gate="0"/>
  <filter protocol="udp" destPort="5060" gate="1"/>
  <filter tos="0b00111000" tosMask="0x3f" gate="2"/>
</filters>
```

Behavior Aggregate Classifier

The `BehaviorAggregateClassifier` module can be used to read the DSCP field from the IP datagram and direct the packet to the corresponding output gate. The DSCP value is the lower six bits of the TypeOfService/TrafficClass field. Core routers usually use this classifier to guide the packet to the appropriate queue.

DSCP values are enumerated in the `dscps` parameter. The first value is for gate `out[0]`, the second for `out[1]`, and so on. If the received packet has a DSCP value that is not enumerated in the `dscps` parameter, it will be forwarded to the `defaultOut` gate.

15.3.5 Meters

Meters classify the packets based on the temporal characteristics of their arrival. The arrival rate of packets is compared to an allowed traffic profile, and packets are either marked as green (in-profile) or red (out-of-profile). Some meters apply more than two conformance levels. For example, in three-color meters, packets that partially conform are classified as yellow.

The allowed traffic profile is usually specified by a token bucket. In this model, a bucket is filled with tokens at a specified rate until it reaches its maximum capacity. When a packet arrives, the bucket is examined. If it contains

at least as many tokens as the length of the packet, then those tokens are removed, and the packet is marked as conforming to the traffic profile. If the bucket contains fewer tokens than needed, it is left unchanged, but the packet is marked as non-conforming.

Meters have two modes: color-blind and color-aware. In color-blind mode, the color assigned by a previous meter does not affect the classification of the packet in subsequent meters. In color-aware mode, the color of the packet cannot be changed to a less conforming color. If a packet is classified as non-conforming by any meter, it is also handled as non-conforming in subsequent meters in the data path.

Important

Meters take into account the length of the IP packet only; L2 headers are omitted from the length calculation. If they receive a packet that is not an IP datagram and does not encapsulate an IP datagram, an error occurs.

TokenBucketMeter

The `TokenBucketMeter` module implements a simple token bucket meter. The module has two outputs: one for green packets and one for red packets. When a packet arrives, the gained tokens are added to the bucket, and the number of tokens equal to the size of the packet are subtracted.

Packets are classified according to two parameters: Committed Information Rate (cir) and Committed Burst Size (cbs), as either green or red.

Green traffic is guaranteed to be under $cir \cdot (t_1 - t_0) + 8 \cdot cbs$ in every $[t_0, t_1]$ interval.

SingleRateThreeColorMeter

The `SingleRateThreeColorMeter` module implements a Single Rate Three Color Meter (RFC 2697). The module has three outputs: green, yellow, and red packets.

Packets are classified according to three parameters: Committed Information Rate (cir), Committed Burst Size (cbs), and Excess Burst Size (ebs), as either green, yellow, or red. The green traffic is guaranteed to be under $cir \cdot (t_1 - t_0) + 8 \cdot cbs$ while the green+yellow traffic is guaranteed to be under $cir \cdot (t_1 - t_0) + 8 \cdot (cbs + ebs)$ in every $[t_0, t_1]$ interval.

TwoRateThreeColorMeter

The `TwoRateThreeColorMeter` module implements a Two Rate Three Color Meter (RFC 2698). The module has three output gates for green, yellow, and red packets.

It classifies packets based on two rates: Peak Information Rate (pir) and Committed Information Rate (cir) and their associated burst sizes (pbs and cbs), as either green, yellow, or red. The green traffic is guaranteed to be under $pir \cdot (t_1 - t_0) + 8 \cdot pbs$ and $cir \cdot (t_1 - t_0) + 8 \cdot cbs$, the yellow traffic is guaranteed to be under $pir \cdot (t_1 - t_0) + 8 \cdot pbs$ in every $[t_0, t_1]$ interval.

15.3.6 Markers

DSCP markers set the codepoint (DSCP field) of packets. The codepoint determines the further processing of the packet in the router or in the core of the DiffServ domain.

The `DscpMarker` module sets the DSCP field (lower six bits of the TypeOfService/TrafficClass field) of IP datagrams to the values specified by the `dscps` parameter. The `dscps` parameter is a space-separated list of codepoints. A different value can be specified for each input gate, where the i^{th} value is used for packets arriving at the i^{th} input gate. If there are fewer values than gates, the last value is used for any extra gates.

The DSCP values are enumerated in the `DSCP.msg` file. You can specify DSCP values using either their names or integer values in the `dscps` parameter.

For example, the following lines are equivalent:

```
**.dscps = "EF 0x0a 0b00001000"
**.dscps = "46 AF11 8"
```

15.4 Compound Modules

15.4.1 AFxyQueue

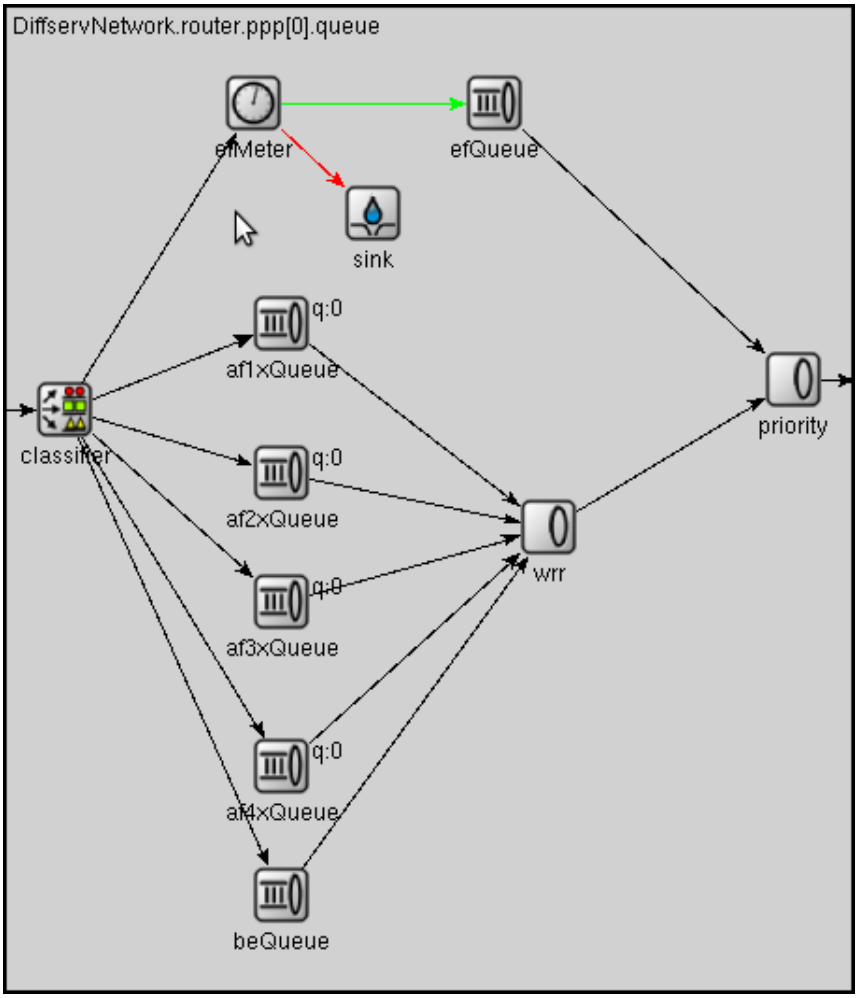
The `AFxyQueue` module is an example queue that implements one class of the Assured Forwarding PHB group (RFC 2597).

Packets with the same AFx class but different drop priorities arrive at the `afx1In`, `afx2In`, and `afx3In` gates. The received packets are stored in the same queue. Before the packet is enqueued, a RED dropping algorithm may decide to selectively drop them, based on the average length of the queue and the RED parameters of the drop priority of the packet.

The afxyMinth, afxyMaxth, and afxyMaxp parameters must have values that ensure that packets with lower drop priorities are dropped with lower or equal probability than packets with higher drop priorities.

15.4.2 DiffservQueue

The `DiffservQueue` is an example queue that can be used in interfaces of DiffServ core and edge nodes to support the AFxy (RFC 2597) and EF (RFC 3246) PHBs.



The incoming packets are first classified according to their DSCP field. DSCP values other than AFxy and EF are handled as best effort (BE).

EF packets are stored in a dedicated queue and served first when a packet is requested. Because they can preempt the other queues, the rate of the EF packets should be limited to a fraction of the bandwidth of the link. This is achieved by metering the EF traffic with a token bucket meter and dropping packets that do not conform to the traffic profile.

There are other queues for AFx classes and BE. The AFx queues use RED to implement three different drop priorities within the class. BE packets are stored in a drop tail queue. Packets from AFxy and BE queues are scheduled by a WRR scheduler, which ensures that the remaining bandwidth is allocated among the classes according to the specified weights.

CHAPTER

SIXTEEN

THE MPLS MODELS

16.1 Overview

Multi-Protocol Label Switching (MPLS) is a "layer 2.5" protocol for high-performance telecommunications networks. MPLS data is directed from one network node to the next using numeric labels instead of network addresses. This avoids complex lookups in a routing table and allows for traffic engineering. The labels identify virtual links (label-switched paths or LSPs, also called MPLS tunnels) between distant nodes, rather than endpoints. The routers that make up a label-switched network are called label-switching routers (LSRs) inside the network ("transit nodes"), and label edge routers (LER) on the edges of the network ("ingress" or "egress" nodes).

A fundamental MPLS concept is that the meaning of the labels used to forward traffic between and through two LSRs must be agreed upon. This common understanding is achieved by using signaling protocols through which one LSR informs another of label bindings it has made. Such signaling protocols are also called label distribution protocols. The two main label distribution protocols used with MPLS are LDP and RSVP-TE.

INET provides basic support for building MPLS simulations. It provides models for the MPLS, LDP, and RSVP-TE protocols and their associated data structures, and preassembled MPLS-capable router models.

16.2 Core Modules

The core modules are:

- `Mpls` implements the MPLS protocol
- `LibTable` holds the LIB (Label Information Base)
- `Ldp` implements the LDP signaling protocol for MPLS
- `RsvpTe` implements the RSVP-TE signaling protocol for MPLS
- `Ted` contains the Traffic Engineering Database
- `LinkStateRouting` is a simple link-state routing protocol
- `RsvpClassifier` is a configurable ingress classifier for MPLS

16.2.1 Mpls

The MPLS protocol is implemented by the `Mpls` module. MPLS is situated between layer 2 and 3, and its main function is to switch packets based on their labels. For that purpose, it relies on the data structure called LIB (Label Information Base). LIB is essentially a table with the following columns: *input-interface*, *input-label*, *output-interface*, *label-operation(s)*.

When receiving a labeled packet from another LSR, MPLS first extracts the incoming interface and incoming label pair, and then looks it up in the local LIB. If a matching entry is found, it applies the prescribed label operations and forwards the packet to the output interface.

The label operations can be one of the following:

- *Push* adds a new MPLS label to a packet. (A packet may contain multiple labels, acting as a stack.) When a normal IP packet enters an LSP, the new label will be the first label on the packet.

- *Pop* removes the topmost MPLS label from a packet. This is typically done at either the penultimate or the egress router.
- *Swap* replaces the topmost label with a new label.

In INET, the local LIB is stored in a `LibTable` module in the router.

When receiving an unlabeled (e.g., plain IPv4) packet, MPLS first determines the forwarding equivalence class (FEC) for the packet using an ingress classifier, and then inserts one or more labels in the packet's newly created MPLS header. The packet is then passed on to the next hop router for the LSP.

The ingress classifier is also a separate module; it is selected depending on the choice of the signaling protocol.

16.2.2 LibTable

The `LibTable` module stores the LIB (Label Information Base), as described in the previous section. The router is expected to have one instance of the `LibTable` module.

LIB is normally filled and maintained by label distribution protocols (RSVP-TE, LDP), but in INET, it is possible to preload it with initial contents.

The `LibTable` module accepts an XML config file whose structure follows the contents of the LIB table. An example configuration:

```
<libtable>
    <libentry>
        <inLabel>203</inLabel>
        <inInterface>ppp1</inInterface>
        <outInterface>ppp2</outInterface>
        <outLabel>
            <op code="pop"/>
            <op code="swap" value="200"/>
            <op code="push" value="300"/>
        </outLabel>
        <color>200</color>
    </libentry>
</libtable>
```

There can be multiple `<libentry>` elements, each describing a row in the table. Columns are given as child elements: `<inLabel>`, `<inInterface>`, etc. The `<color>` element is optional, and it only exists to be able to color LSPs on the GUI. It is not used by the protocols.

16.2.3 Ldp

The `Ldp` module implements the Label Distribution Protocol (LDP). LDP is used to establish LSPs in an MPLS network when traffic engineering is not required. LDP establishes LSPs that follow the existing IP routing table, and is particularly well suited for establishing a full mesh of LSPs between all routers on the network.

LDP relies on the underlying routing information provided by a routing protocol to forward label packets. The router's forwarding information base, or FIB, determines the hop-by-hop path through the network.

In INET, the `Ldp` module takes routing information from the `Ted` module. The `Ted` instance in the network is filled and maintained by a `LinkStateRouting` module. Unfortunately, it is currently not possible to use other routing protocol implementations such as `Ospfv2` in conjunction with `Ldp`.

When `Ldp` is used as the signaling protocol, it also serves as the ingress classifier for `Mpls`.

16.2.4 Ted

The `Ted` module contains the Traffic Engineering Database (TED). In INET, `Ted` contains a link-state database, including reservations for each link by RSVP-TE.

16.2.5 LinkStateRouting

The `LinkStateRouting` module provides a simple link-state routing protocol. It uses `Ted` as its link-state database. Unfortunately, the `LinkStateRouting` module cannot operate independently; it can only be used inside an MPLS router.

16.2.6 RsvpTe

The `RsvpTe` module implements RSVP-TE (Resource Reservation Protocol – Traffic Engineering) as the signaling protocol for MPLS. RSVP-TE handles bandwidth allocation and allows traffic engineering across an MPLS network. Like LDP, RSVP uses discovery messages and advertisements to exchange LSP path information between all hosts. However, whereas LDP is restricted to using the configured IGP's shortest path as the transit path through the network, RSVP can take into consideration network constraint parameters such as available bandwidth and explicit hops. RSVP uses a combination of the Constrained Shortest Path First (CSPF) algorithm and Explicit Route Objects (EROs) to determine how traffic is routed through the network.

When `RsvpTe` is used as the signaling protocol, `Mpls` requires a separate ingress classifier module, which is usually a `RsvpClassifier`.

The `RsvpTe` module allows LSPs to be specified statically in an XML config file. An example `traffic.xml` file:

```
<sessions>
    <session>
        <endpoint>host3</endpoint>
        <tunnel_id>1</tunnel_id>
        <paths>
            <path>
                <lspid>100</lspid>
                <bandwidth>100000</bandwidth>
                <route>
                    <node>10.1.1.1</node>
                    <lnode>10.1.2.1</lnode>
                    <node>10.1.4.1</node>
                    <node>10.1.5.1</node>
                </route>
                <permanent>true</permanent>
                <color>100</color>
            </path>
        </paths>
    </session>
</sessions>
```

In the route, <node> stands for a strict hop, and <lnode> stands for a loose hop.

Paths can also be set up and torn down dynamically with `ScenarioManager` commands (see chapter *Scenario Scripting*). `RsvpTe` understands the <add-session> and <del-session> `ScenarioManager` commands. The contents of the <add-session> element can be the same as the <session> element for the `traffic.xml` above. The <del-session> element syntax is also similar, but only <endpoint>, <tunnel_id>, and <lspid> need to be specified.

The following is an example `scenario.xml` file:

```
<scenario>
    <at t="2">
        <add-session module="LSR1.rsvp">
            <endpoint>10.2.1.1</endpoint>
            <tunnel_id>1</tunnel_id>
            <paths>
                ...
            </paths>
```

(continues on next page)

(continued from previous page)

```
        </add-session>
    </at>
    <at t="2.4">
        <del-session module="LSR1.rsvp">
            <endpoint>10.2.1.1</endpoint>
            <tunnel_id>1</tunnel_id>
            <paths>
                <path>
                    <lspid>100</lspid>
                </path>
            </paths>
        </del-session>
    </at>
</scenario>
```

16.3 Classifier

The `RsvpClassifier` module implements an ingress classifier for `Mpls` when using `RsvpTe` for signaling. The classifier can be configured with an XML config file.

```
**.classifier.config = xmldoc("fectable.xml");
```

An example `fectable.xml` file:

```
<fectable>
    <fecentry>
        <id>1</id>
        <destination>host5</destination>
        <source>host1</source>
        <tunnel_id>1</tunnel_id>
        <lspid>100</lspid>
    </fecentry>
</fectable>
```

16.4 MPLS-Enabled Router Models

INET provides the following preassembled MPLS routers:

- `LdpMplsRouter` is an MPLS router with the LDP signaling protocol
- `RsvpMplsRouter` is an MPLS router with the RSVP-TE signaling protocol

CHAPTER

SEVENTEEN

POINT-TO-POINT LINKS

17.1 Overview

For simulating wired point-to-point links, the INET Framework contains a minimal implementation of the PPP protocol and a corresponding network interface module.

- `Ppp` is a simple module that performs encapsulation of network datagrams into PPP frames and decapsulation of the incoming PPP frames. It can be connected to the network layer directly or can be configured to get the outgoing messages from an output queue. The module collects statistics about the transmitted and dropped packages.
- `PppInterface` is a compound module that complements the `Ppp` module with an output queue. It implements the `IWiredInterface` interface. Input and output hooks can be configured for further processing of the network messages.

PPP (RFC 1661) is a complex protocol which, in addition to providing a method for encapsulating multi-protocol datagrams, also contains control protocols for establishing, configuring, and testing the data-link connection (LCP) and for configuring different network-layer protocols (NCP).

The INET implementation only covers encapsulation and decapsulation of data into PPP frames. Control protocols, which do not have a significant effect on the link's capacity and latency during normal link operation, are not simulated. In addition, header field compressions (PFC and ACFC) are also not supported, so a simulated PPP frame always contains 1-byte Address and Control fields and a 2-byte Protocol field.

17.2 The PPP module

The PPP module receives packets from the upper layer in the `netwIn` gate, adds a `PppHeader`, and sends it to the physical layer through the `phys` gate. The packet with `PppHeader` is received from the `phys` and sent to the upper layer immediately through the `netwOut` gate.

Incoming datagrams are waiting in a queue if the line is currently busy. In routers, PPP relies on an external queue module (implementing `IPacketQueue`) to model finite buffer, implement QoS and/or RED, and requests packets from this external queue one-by-one. The name of this queue is given as the `queueModule` parameter.

In hosts, no such queue is used, so `Ppp` contains an internal queue to store packets waiting for transmission. Conceptually the queue is of infinite size, but for better diagnostics one can specify a hard limit in the `packetCapacity` parameter – if this is exceeded, the simulation stops with an error.

The module can be used in simulations where the nodes are connected and disconnected dynamically. If the channel between the PPP modules is down, the messages received from the upper layer are dropped (including the messages waiting in the queue). When the connection is restored, it will poll the queue and transmit the messages again.

The PPP module registers itself in the interface table of the node. The `mtu` of the entry can be specified by the `mtu` module parameter. The module checks the state of the physical link and updates the entry in the interface table.

17.3 PppInterface

`PppInterface` is a compound module that implements the `IWiredInterface` interface. It contains a `Ppp` module and a passive queue for the messages received from the network layer.

The queue type is specified by the `typename` parameter of the queue submodule. It can be set to `PacketQueue` or to a module type implementing the `IPacketQueue` interface. There are implementations with QoS and RED support.

In typical use of the `Ppp` module, it is augmented with other nodes that monitor the traffic or simulate package loss and duplication. The `PppInterface` module abstracts that usage by adding `IHook` components to the network input and output of the `Ppp` component. Any number of hooks can be added by specifying the `numOutputHooks` and `numInputHooks` parameters and the types of the `outputHook` and `inputHook` components. The hooks are chained in their numeric order.

CHAPTER

EIGHTEEN

THE ETHERNET MODEL

18.1 Overview

Ethernet is the most popular wired LAN technology nowadays, and its use is also growing in metropolitan area and wide area networks. Since its introduction in 1980, Ethernet data transfer rates have increased from the original 10Mb/s to the latest 400Gb/s. Originally, the technology has changed from using coaxial cables and repeaters to using unshielded twisted-pair cables with hubs and switches. Today, switched Ethernet is prevalent, and most links operate in full duplex mode. Support for all major Ethernet technologies and device types is provided by the INET Framework.

In Ethernet networks containing multiple switches, broadcast storms are prevented by the use of a spanning tree protocol (STP, RSTP) that disables selected links to eliminate cycles from the topology. STP and RSTP are supported in the Ethernet switch models in INET.

18.2 Nodes

Several node models can be used in an Ethernet network:

- Node models such as `StandardHost` and `Router` are Ethernet-capable.
- `EthernetSwitch` models an Ethernet switch, which is a multiport bridging device.
- `WireJunction` can model the coaxial cable (10BASE2 or 10BASE5 network segments) on legacy Ethernet networks, or an Ethernet hub/multiport repeater.
- `EthernetHost` is a sample node that can be used to generate "raw" Ethernet traffic.

18.2.1 Ethernet Switch

`EthernetSwitch` models an Ethernet switch. Ethernet switches play an important role in modern Ethernet LANs. Unlike passive hubs and repeaters that work in the physical layer, the switches operate in the data link layer and relay frames between the connected subnets.

In modern Ethernet LANs, each node is connected to the switch directly by full duplex lines, so no collisions are possible. In this case, the CSMA/CD is not needed and the channel utilization can be high.

The `duplexMode` parameters of the MACs must be set according to the medium connected to the port; if collisions are possible (it's a bus or hub) it must be set to false, otherwise, it can be set to true. By default, it uses a half-duplex MAC with CSMA/CD.

18.2.2 Ethernet Hub

The `WireJunction` can model an Ethernet hub. Ethernet hubs are simple broadcast devices. Messages arriving on a port are regenerated and broadcast to every other port.

The connections connected to the hub must have the same data rate. Cable lengths should be reflected in the delays of the connections.

18.2.3 Ethernet Bus

The `WireJunction` component can also model a connection to a common coaxial cable found in early Ethernet LANs. Network nodes are attached to the `WireJunction` via a `DatarateChannel`. The `WireJunction` modules are connected to each other via `DatarateChannel` as well. When a node sends a signal, it will propagate along the cable in both directions at the given propagation speed.

The speed of the connection can be set on the datarate channels; all connected channels must have the same speed.

18.3 The Physical Layer

Stations on Ethernet networks are connected by coaxial, twisted pair or fiber cables. (Coaxial only has historical importance, but is supported by INET anyway.) There are several cable types specified in the standard.

In the INET framework, the cables are represented by connections. The connections used in Ethernet LANs must be derived from ned::`DatarateChannel` and should have their `delay` and `datarate` parameters set. The delay parameter can be used to model the distance between the nodes. The datarate parameter can have four values:

- 10Mbps (classic Ethernet)
- 100Mbps (Fast Ethernet)
- 1Gbps (Gigabit Ethernet, GbE)
- 10Gbps (10 Gigabit Ethernet, 10GbE)
- 40Gbps (40 Gigabit Ethernet, 40GbE)
- 100Gbps (100 Gigabit Ethernet, 100GbE)

There is currently no support for 200Gbps and 400Gbps Ethernet.

`Eth10M`, `Eth100M`, `Eth1G`, `Eth10G`, `Eth40G`, `Eth100G`

18.4 Ethernet Interface

The `EthernetInterface` compound module implements the `IWiredInterface` interface. Complements `EthernetCsmaMacPhy` and `EthernetEncapsulation` with an output queue for QoS and RED support. It also has configurable input/output filters as `IHook` components similarly to the `PppInterface` module.

The Ethernet MAC (Media Access Control) layer transmits the Ethernet frames on the physical media. This is a sublayer within the data link layer. Because encapsulation/decapsulation is not always needed (e.g., switches do not do encapsulation/decapsulation), it is implemented in separate modules (e.g., `EthernetEncapsulation`) that are part of the LLC layer.

Nowadays, almost all Ethernet networks operate using full-duplex point-to-point connections between hosts and switches. This means that there are no collisions, and the behavior of the MAC component is much simpler than in classic Ethernet that used coaxial cables and hubs. Two MAC modules for Ethernet are provided by the INET framework: the `EthernetMacPhy` is simpler to understand and easier to extend, because it supports only full-duplex connections. The `EthernetCsmaMacPhy` module implements the full MAC functionality including CSMA/CD, it can operate in both half-duplex and full-duplex mode.

18.5 Components

The following components are present in the model:

- `EthernetMacPhy`
- `EthernetCsmaMacPhy`
- `EthernetEncapsulation`
- `MacRelayUnit`
- `MacForwardingTable`

- Ieee8021dRelay

18.5.1 EthernetMacPhy

From the two MAC implementations, EthernetMacPhy is the simpler one, it operates only in full-duplex mode (its duplexEnabled parameter fixed to true in its NED definition). This module does not need to implement CSMA/CD, so there is no collision detection, retransmission with exponential backoff, carrier extension, and frame bursting.

18.5.2 EthernetCsmaMacPhy

Ethernet MAC layer implementing CSMA/CD. It supports both half-duplex and full-duplex operations; in full-duplex mode, it behaves as EthernetMacPhy. In half-duplex mode, it detects collisions, sends jam messages, and retransmits frames upon collisions using the exponential backoff algorithm. In Gigabit Ethernet networks, it supports carrier extension and frame bursting. Carrier extension can be turned off by setting the carrierExtension parameter to false.

18.5.3 EthernetEncapsulation

The EthernetEncapsulation module performs Ethernet II or Ethernet with SNAP encapsulation/decapsulation.

18.5.4 MacRelayUnit

INET framework Ethernet switches are built from IMacRelayUnit components. Each relay unit has N input and output gates for sending/receiving Ethernet frames. They should be connected to EthernetInterface modules.

The relay unit holds a table for the destination address -> output port mapping in a MacForwardingTable module. When the relay unit receives a data frame, it updates the table with the source address -> input port.

If the destination address is not found in the table, the frame is broadcasted. The frame is not sent to the same port it was received from, because then the target should already have received the original frame.

A simple scheme for sending PAUSE frames is built-in (although users will probably change it). When the buffer level goes above a high watermark, PAUSE frames are sent on all ports. The watermark and the pause time is configurable; use zero values to disable the PAUSE feature.

18.5.5 MacForwardingTable

The MacForwardingTable module stores the mapping between ports and MAC addresses. Entries are deleted if their age exceeds a certain limit.

If needed, address tables can be pre-loaded from text files at the beginning of the simulation; this is controlled by the forwardingTableFile module parameter. In the file, each line contains a literal 0 (reserved for VLAN id), a hexadecimal MAC address and a decimal port number, separated by tabs. Comment lines beginning with "#" are also allowed:

```
0    01 ff ff ff ff    0
0    00-ff-ff-ee-d1    1
0    0A:AA:BC:DE:FF    2
```

Entries are deleted if their age exceeds the duration given as the agingTime parameter.

18.5.6 Ieee8021dRelay

Ieee8021dRelay is a MAC relay unit that should be used instead of MacRelayUnit when STP or RSTP is needed.

18.5.7 Stp

The Stp module type implements Spanning Tree Protocol (STP). STP is a network protocol that builds a loop-free logical topology for Ethernet networks. The basic function of STP is to prevent bridge loops and the broadcast radiation that results from them.

STP creates a spanning tree within a network of connected layer-2 bridges and disables those links that are not part of the spanning tree, leaving a single active path between any two network nodes.

18.5.8 Rstp

`Rstp` implements Rapid Spanning Tree Protocol (RSTP), an improved version of STP. RSTP provides significantly faster recovery in response to network changes or failures.

18.6 Implemented Standards

The Ethernet model operates according to the following standards:

- Ethernet: IEEE 802.3-1998
- Fast Ethernet: IEEE 802.3u-1995
- Full-Duplex Ethernet with Flow Control: IEEE 802.3x-1997
- Gigabit Ethernet: IEEE 802.3z-1998

CHAPTER

NINETEEN

ETHERNET 10BASE-T1S

This chapter outlines the INET Framework's implementation of select features from the IEEE 802.3cg-2019 standard. Specifically, it covers 10BASE-T1S in both point-to-point and multidrop topologies, utilizing the Physical Layer Collision Avoidance (PLCA) protocol.

19.1 Overview

The Ethernet 10BASE-T1S model is a distinct implementation from the INET Framework's default Ethernet model. Although the default The 10BASE-T1S model re-implements the default model's CSMA/CD functionality to have separate CSMA/CD MAC and CSMA/CD PHY layers. This separation is essential for incorporating the Physical Layer Collision Avoidance (PLCA) protocol between the MAC and PHY layers. The new model is geared towards multidrop link topologies and it offers the following modules for this purpose:

- `EthernetPlcaHost` models a network node suitable for use in multidrop links
- `EthernetPlcaInterface` models a network interface suitable for use in multidrop links
- `EthernetPlca` implements the Ethernet Physical Layer Collision Avoidance (PLCA) protocol
- `EthernetCsmaMac` implements the Ethernet CSMA/CD MAC protocol suitable for use with PLCA
- `EthernetCsmaPhy` implements the Ethernet CSMA/CD PHY protocol suitable for use with PLCA
- `EthernetMultidropLink` implements the single twisted-pair cable suitable for use in multidrop links

Additionally, the model incorporates modules non-specific to 10BASE-T1S:

- `WireJunction` represents a universal wiring hub suitable for use in multidrop links

19.2 Point-to-point links

For the Ethernet 10BASE-T1S point-to-point communication mode the default Ethernet model can be used, more specifically the `EthernetLink` channel and the `EthernetInterface` module. In the network interface, the full-duplex `EthernetMacPhy` module can be employed.

19.3 Multidrop links

To construct an Ethernet 10BASE-T1S multidrop link, connect multiple `EthernetPlcaHost` modules together. Utilize `EthernetMultidropLink` channels and `WireJunction` modules to establish the network topology. Set the cable length for each connection to define the signal propagation delay.

For mixed networks, you can incorporate other network node modules like `StandardHost`, `EthernetSwitch`, or `TsnSwitch`. Replace their network interface directed towards the multidrop link with an `EthernetPlcaInterface` module. Make sure every interface connected to the multidrop link utilizes the `EthernetPlcaInterface` module.

19.4 Result analysis

The 10BASE-T1S modules offer various statistics useful for evaluating network node performance on a multidrop link. For a comprehensive list of statistics, check the `EthernetCsmaMac`, `EthernetPlca`, and `EthernetCsmaPhy` modules.

Key statistics are:

- queue length
- MAC state machine state
- PLCA transmit opportunity ID
- PLCA control state machine state
- PLCA data state machine state
- PHY state machine state
- PHY transmitted signal type
- PHY received signal type

Understanding the operation of a multidrop network interface can become quite complex. Consider creating a chart with multiple subplots and a shared X-axis, displaying various statistics simultaneously. To create such a chart:

- select the desired statistics on the 'Browse data' page of the result analysis tool
- choose 'Plot using line chart on separate axis with Matplotlib' from the context menu
- open the chart configuration dialog, go to the 'Line' tab and enable 'Display enums as colored strips'.

Note

For long running simulations, the recorded statistical results may grow large, causing chart performance to degrade, especially for the colored strips displaying state machine states. To address this issue, open the chart configuration dialog and specify a start/end time on the 'Input' tab to narrow down the data to the relevant portion.

CHAPTER

TWENTY

TIME-SENSITIVE NETWORKING

This chapter describes the part of the INET Framework that implements a subset of the IEEE standards related to Time-Sensitive Networking (TSN). The supported TSN features include, among others: time synchronization, per-stream filtering and policing, scheduling and traffic shaping, frame replication and elimination, frame preemption, cut-through switching, automatic network configuration for failure protection, stream redundancy, and gate scheduling.

The above TSN features are implemented in several modules separate from other standard Ethernet functionality. Some of the modules presented in this chapter are specific to the IEEE standards, but several of them are more generic and they are simply reused to achieve the required functionality.

However, it must be noted that several TSN features are only partially supported. The following sections describe the individual TSN features in detail.

20.1 Devices

TSN networks require additional functionality compared to standard Ethernet. To facilitate the configuration of the TSN features, INET provides several TSN specific network devices. These are derived from the basic INET network nodes and provide several additional TSN specific parameters. The additional parameters mostly determine the internal module structure of the TSN specific network nodes, and also further parameterize the basic network node submodules. The usage of these network nodes is completely optional, and it is possible to combine the TSN modules in other ways. However, it is advisable to start making TSN simulations using them.

- `TsnClock` models a hardware device that exclusively operates as a gPTP master node for time synchronization
- `TsnDevice` models a TSN specific end device, capable of running multiple applications and all other supported TSN features
- `TsnSwitch` models an Ethernet switch that is capable of all supported TSN features

Using the TSN specific network nodes and having the TSN specific features enabled is just the first step for using Time-Sensitive Networking. Most TSN features require additional configuration in the corresponding modules.

20.2 Time Synchronization

This section describes the modules that implement a subset of the IEEE 802.1AS standard titled "Timing and Synchronization for Time-Sensitive Applications" (IEEE 802.1AS-2020). There are two main components for this TSN feature: clock modules that keep track of time in the individual network nodes, and time synchronization protocol modules that synchronize these clocks. All required modules are already included in the TSN specific network nodes.

If the default TSN specific network nodes are not used, then the following modules can still be used to keep track of time in the network nodes.

- `OscillatorBasedClock` models a clock that has a potentially drifting oscillator
- `SettableClock` extends the previous model with the capability of setting the clock time

Similarly to the above, the following gPTP time synchronization related protocol modules and network nodes can also be used to build time synchronization in a network:

- `Gptp` the gPTP time synchronization protocol
- `GptpBridge` models a gPTP time synchronization bridge network node
- `GptpEndstation` models a gPTP time synchronization end station network node
- `GptpMaster` models a gPTP time synchronization master network node
- `GptpSlave` models a gPTP time synchronization slave network node

In order to implement node failure (e.g. master clock) and link failure (e.g. between gPTP bridges) protection, multiple time synchronization domains are required. These time domains operate independently of each other, and it's up to the clock user modules of each network node to decide which clock they are using. Typically, they use the active clock of the `MultiClock`, and there has to be some means of changing the active clocks when failover happens. The following modules can be used to implement multiple time domains:

- `MultiClock` contains several subclocks for the different time domains
- `MultiDomainGptp` contains several gPTP submodules for the different time domains

The following parameters can be used to enable the gPTP time synchronization in various predefined network nodes:

- `hasTimeSynchronization` parameter enables time synchronization in TSN specific network nodes
- `hasGptp` parameter enables the gPTP time synchronization protocol in gPTP specific network nodes

20.3 Per-stream Filtering and Policing

This section describes the modules that implement a subset of the functionality of the IEEE 802.1Q standard that was originally introduced by the Per-Stream Filtering and Policing (IEEE 802.1Qci-2017) amendment.

The simplest module for IEEE 802.1Q per-stream filtering and policing is the `SimpleIeee8021qFilter` compound module. This module combines several submodules: a packet classifier at the input, a packet multiplexer at the output, and one packet meter, one packet filter, and one packet gate per stream. Each one of the latter per-stream 3 modules is optional.

When a packet arrives at the input of the `SimpleIeee8021qFilter`, it first gets classified into one of the filtering and policing submodule paths. Then the packet meter measures the packet as part of the packet stream that has been seen so far and attaches the result of the measurement. The result may be as simple as a label on the packet. After the metering, the packet filter checks if the packet matches the required conditions and either lets the packet go through or drops it. Finally, the packet gate allows for the automatic time based or programmatic control of the packet passing through the selected path of the policing module. Packets are never enqueued in the `SimpleIeee8021qFilter`, they either pass through or get dropped immediately.

Note that any of the `SimpleIeee8021qFilter` default submodules can be replaced with other variants. Moreover, other more complicated internal structures are also possible. This is especially the case when the packet meters are replaced with token bucket classifiers as described below.

As the first step, the default policing process starts with a packet classifier, module, the `StreamClassifier` by default that classifies packets based on the attached stream information. This classifier simply maps stream names to output gate indices. Please note that the stream decoding and identification process is not part of the `SimpleIeee8021qFilter`.

In the second step, the default policing process continues with a packet meter module, the `DualRateThreeColorMeter` by default, that labels the packets as either green, yellow, or red based on the committed and excess information rate and the committed and excess burst size parameters.

The most commonly used packet meters for per-stream filtering and policing are:

- `SingleRateTwoColorMeter` labels packets as green or red based on CIR and CBS parameters
- `SingleRateThreeColorMeter` labels packets as green, yellow, or red based on CIR, CBS, and EBS parameters

- `DualRateThreeColorMeter` labels packets as green, yellow, or red based on CIR, CBS, EIR, and EBS parameters

The above modules are based on the following generic token bucket meter modules:

- `TokenBucketMeter` contains a single token bucket and labels packets as one of 2 labels
- `MultiTokenBucketMeter` contains an overflowing chain of N token buckets and labels packets with one of N+1 labels

Different packet meter modules can also be used by replacing the default packet meter submodules of `SimpleIeee8021qFilter`. See the inet.queueing.meter NED package for alternatives.

In the third step, the default per-stream filtering and policing process continues with a packet filter module, the `LabelFilter` by default, that drops the red packets and lets through the green and yellow ones by default. Of course, different packet filter modules can also be used by replacing the default filter submodules of `SimpleIeee8021qFilter`. See the inet.queueing.filter NED package for alternatives.

Finally, the default policing process finishes by merging the per-stream filtering and policing paths into a single output gate by using the generic `PacketMultiplexer` module. There's no need to prioritize between the per-stream paths here because the packets pass through in zero simulation time.

Different per-stream filtering and policing compound modules can also be created by combining the existing queueing and protocol element modules of the INET Framework. For example, instead of the packet meter modules, the token bucket-based packet classifier modules give more freedom in terms of the module structure. See the inet.queueing NED package for more modules.

The most commonly used packet classifiers for per-stream filtering and policing are:

- `SingleRateTwoColorClassifier` classifies packets to 2 output gates based on CIR and CBS parameters
- `SingleRateThreeColorClassifier` classifies packets to 3 output gates based on CIR, CBS, and EBS parameters
- `DualRateThreeColorClassifier` classifies packets to 3 output gates based on CIR, CBS, EIR, and EBS parameters

The above modules are derived from the generic token bucket classifier modules. These modules can also be used on their own and combined in many different ways with all the other queueing modules to achieve the desired per-stream filtering and policing.

- `TokenBucketClassifier` contains a single token bucket and classifies packets to 2 output gates
- `MultiTokenBucketClassifier` contains an overflowing chain of N token buckets and classifies packets to the N+1 output gates

There is also a more complex per-stream filtering and policing module, called the `Ieee8021qFilter`. This module is more similar to the architecture that is present in the IEEE 802.1Q standard. The `Ieee8021qFilter` also combines several submodules but in a slightly different way than the `SimpleIeee8021qFilter`. The most important difference is that this module can be mostly configured through a single streamFilterTable parameter.

The TSN specific network node `TsnDevice` and `TsnSwitch` have a special parameter called `hasIngressTrafficFiltering` which can be used to enable the traffic filtering and policing in the network node architecture. Of course, these modules can also be used in other ways.

20.4 Scheduling and Traffic Shaping

This section describes the modules that implement a subset of the functionality of the IEEE 802.1Q standard that was originally introduced by the Enhancements for Scheduled Traffic (IEEE 802.1Qbv-2015) amendment.

The traffic shaping architecture is part of the queue submodule of the MAC layer in the network interface. Currently, three different packet shaper algorithms are supported: the credit-based shaper, the time-aware shaper, and the asynchronous shaper. In order to configure the network interface to use traffic shaping, the queue submodule must be replaced with either the `GatingPriorityQueue` or the `PriorityShaper` compound modules. Both contain a packet classifier to differentiate between the traffic categories and a priority packet scheduler that prefers higher

priority traffic categories over lower priority ones. The difference is in the structure of the other submodules that form the shapers.

The credit-based shaper is implemented in the `CreditBasedShaper` module using a standard `PacketQueue` and a special-purpose `Ieee8021qCreditBasedGate` submodule. The latter module keeps track of the available credits for the given traffic category and allows or forbids the transmission of packets.

The time-aware shaper is implemented in the `TimeAwareShaper` compound module that uses a standard `PacketQueue` and a special-purpose `PeriodicGate`. The latter module has parameters to control the gate schedule that determines the periodic open and gate.

The asynchronous shaper is, in part, implemented in the `AsynchronousShaper` compound module. This shaper is somewhat more complicated than the previous two because it also contains submodules that are part of the ingress per-stream filtering module in the bridging layer. These are the `EligibilityTimeMeter` and the corresponding `EligibilityTimeFilter` submodules. The first is responsible for calculating the transmission eligibility time for incoming packets, the latter is responsible for dropping packets which are considered too old for transmission. The shaper in the network interface queue contains two additional submodules called `EligibilityTimeQueue` and `EligibilityTimeGate`. The former is responsible for sorting the frames according to the transmission eligibility time, the latter is a gate that is open only if the transmission eligibility time of the first frame of the queue is greater than the current time.

The TSN specific network node `TsnDevice` and `TsnSwitch` have a special parameter called `hasEgressTrafficShaping` which can be used to enable the traffic shaping in the network node architecture. Of course, these modules can also be used in other ways.

20.5 Frame Replication and Elimination

This section describes the modules that implement a subset of the functionality of the IEEE 802.1CB standard titled "Frame Replication and Elimination for Reliability" (IEEE 802.1CB-2017).

Frame Replication and Elimination for Reliability (FRER) is a mechanism that provides seamless redundancy for time-sensitive networking applications. The core idea is to protect critical data streams against link failures and packet loss by replicating frames at strategic points in the network and sending them along multiple disjoint paths toward the destination. At merge points and at the destination, duplicate frames are identified using sequence numbers and eliminated.

FRER provides seamless redundancy with zero packet loss during link or node failures, making it ideal for critical industrial control systems, automotive networks, and other time-sensitive applications. From the application's perspective, frame replication is completely transparent - packet delivery continues uninterrupted even during network failures, with the application receiving each packet exactly once.

20.5.1 Key FRER Concepts

Stream Identification

Each packet flow is assigned to a named stream. Stream identification uses packet filters (typically matching MAC addresses, VLAN IDs, or other header fields) to determine which stream a packet belongs to. This allows the network to apply specific replication and elimination policies to different traffic flows.

Sequence Numbering

To enable duplicate detection and elimination, each frame in a stream is assigned a unique sequence number at the source. This sequence number is typically carried in the IEEE 802.1 R-Tag header. The sequence numbers allow downstream merge points and the destination to identify and discard duplicate frames, ensuring each frame is forwarded to the application only once.

Stream Encoding and Decoding

Streams are encoded into VLAN tags as they traverse the network. This allows switches to forward frames based on standard VLAN forwarding tables while maintaining stream identity. At each hop, incoming frames are decoded to determine their stream identity, processed by the FRER mechanisms, and then encoded again for the next hop. The encoding/decoding process maps between stream names (internal identifiers) and VLAN IDs (wire format).

Stream Splitting (Replication)

At replication points in the network, a single input stream is duplicated into multiple output streams, each sent on a different path. This creates redundancy - if one path fails, other copies can still reach the destination. Replication typically occurs at strategic points such as the first switch after the source or at network nodes where multiple disjoint paths diverge. Each replicated stream may be encoded with different VLAN IDs for path separation.

Stream Merging (Elimination)

At merge points in the network, multiple input streams are combined into a single output stream. The merger uses sequence numbers to detect and eliminate duplicate frames, forwarding only the first copy of each frame that arrives. Subsequent duplicates are discarded. Merging occurs wherever redundant paths converge, as well as at the final destination where all remaining copies are eliminated before delivery to the application.

Transparent Operation

From the application's perspective, FRER is completely transparent. The application sends packets normally, and the bridging layer handles all stream identification, replication, encoding, and elimination automatically. The application receives each packet exactly once, regardless of how many copies were transmitted through the network or which paths they took.

20.5.2 FRER Modules

The relevant modules are all part of the `BridgingLayer` compound module that resides between the network layer and link layer protocols. This compound module also contains other functionality such as frame forwarding. There are four relevant submodules, each one implements a very specific part of frame replication.

The first part deals with stream identification and is implemented in the `StreamIdentifierLayer` module and its `StreamIdentifier` submodule. This module is only useful in network nodes which produce application traffic themselves. The stream identifier module is responsible for assigning a stream name for outgoing packets by looking at their contents and metadata. For example, packets can be identified by the destination MAC address and PCP request tags. Since at this point the packets don't yet contain any layer 2 header, the decision can be based on the attached request tags that will be later turned into packet headers.

The second layer handles incoming stream merging and outgoing stream splitting. This layer is called the `StreamRelayLayer` and contains two submodules called `StreamMerger` and `StreamSplitter`. The former is responsible for merging incoming member streams into a single stream and removing duplicate frames. The latter is responsible for splitting outgoing streams into potentially several member streams.

The third part deals with ingress and egress stream filtering and is implemented in the `StreamFilterLayer` module that contains one submodule for both directions. This part is not strictly necessary for frame replication. Most often only the ingress filtering submodule is used as described in the previous section.

The last layer handles incoming packet decoding and outgoing packet encoding. This module is called the `StreamCoderLayer` and it contains two submodules: the `StreamDecoder` and the `StreamEncoder`. The former handles the stream decoding of incoming packets by checking the attached indication tags. The latter deals with the encoding of outgoing packets by attaching the necessary request tags.

The TSN specific network node `TsnDevice` and `TsnSwitch` have a special parameter called `hasStreamRedundancy` which can be used to enable frame replication in the network node architecture. Of course, these modules can also be used in other ways.

20.6 Frame Preemption

This section describes the modules that implement a subset of the functionality of the IEEE 802.1Q standard that was originally introduced by the Frame Preemption (IEEE 802.1Qbu) amendment.

Frame preemption requires the network interface to be able to interrupt an ongoing transmission and switch to the transmission of a higher priority frame. This behavior is implemented in special MAC and PHY layer modules that use packet streaming in the network interface. This is in contrast with the default behavior where modules pass packets around as a whole.

- `EthernetPreemptingMacLayer` models an Ethernet MAC layer that contains multiple MAC sublayers to allow the preemption of background traffic

- `EthernetPreemptingPhyLayer` models a PHY layer that allows the preemption of an ongoing transmission

The TSN specific network nodes `TsnDevice` and `TsnSwitch` have a special parameter called `hasFramePreemption` which can be used to enable frame preemption in the network interfaces. Of course, these modules can also be used in other ways.

20.7 Cut-through Switching

The default store and forward mechanism in Ethernet switches greatly influences the end-to-end latency of application traffic. This effect can be overcome and drastically reduced by using cut-through switching. This method starts forwarding the incoming frame before the whole frame has been received, usually right after the reception of the MAC header. However, cut-through switching is not a standard mechanism, and several variants are in operation.

INET provides the following modules related to cut-through switching:

- `EthernetCutthroughInterface` models an Ethernet interface that contains a special cut-through layer between the MAC and PHY layers that in certain circumstances allows the direct forwarding of frames from the incoming network interface to the outgoing
- `EthernetCutthroughLayer` models the cut-through layer with direct connections to other cut-through interfaces inside the same network node
- `EthernetCutthroughSource` models the source of the cut-through forwarding inside the network interface
- `EthernetCutthroughSink` models the sink of the cut-through forwarding inside the network interface

Surprisingly, cut-through switch also has to be enabled in the end devices because the receiving switch has to be notified both at the start and at the end of the frame.

The TSN specific network nodes `TsnDevice` and `TsnSwitch` have a special parameter called `hasCutthroughSwitching` which can be used to enable cut-through switching in the network interfaces. Of course, these modules can also be used in other ways.

20.8 Automatic Network Configuration

Configuring the features of Time-Sensitive Networking in a complex network that contains many applications with different traffic requirements is a difficult and error-prone task. To facilitate this task, INET provides three types of network-level configurators:

- gate scheduling configurators are capable of configuring the gate control lists (i.e., periodic open/close states) for all traffic classes in all network interfaces based on packet length, packet interval, and maximum latency parameters
- stream redundancy configurators are capable of configuring the stream merging and stream splitting modules as well as the stream identification in all network nodes to form the desired redundant streams for each application traffic
- failure protection configurators are capable of using the previous two to achieve the desired link and node failure protections for all streams in the network based on the set of failure cases

All other network-level configurators such as the `Ipv4NetworkConfigurator` or the `MacForwardingTableConfigurator` can also be used.

There are several different automatic gate scheduling configurators having different capabilities:

- `EagerGateScheduleConfigurator` eagerly allocates time slots in the order of increasing traffic priority
- `Z3GateScheduleConfigurator` uses a SAT solver to fulfill the traffic constraints all at once
- `TSNschedGateScheduleConfigurator` uses a state-of-the-art external tool called TSNsched that is available at https://github.com/ACassimiro/TSNsched

There is only one stream redundancy configurator:

- `StreamRedundancyConfigurator` configures stream splitting, stream merging and stream filtering in all network nodes

Currently, there is only one failure protection configurator:

- `FailureProtectionConfigurator` configures the gate scheduling and the stream redundancy configurators to provide protection against the specified link and node failures

All of these configurators automatically discover the network topology and then taking into account their own independent configuration they compute the necessary parameters for the individual underlying modules and configure them. However, anything they can do, can also be done from INI files manually, and the result can also be seen at the configured module parameters in the runtime user interface.

CHAPTER

TWENTYONE

THE 802.11 MODEL

21.1 Overview

IEEE 802.11 a.k.a. WiFi is the most widely used and universal wireless networking standard. Specifications are updated every few years, adding more features and ever-increasing bit rates.

In INET, nodes become WiFi-enabled by adding an `Ieee80211Interface` to them. (As mentioned earlier, `WirelessHost` and `AdhocHost` already contain one in their default configuration.) APs are represented with the `AccessPoint` node type. WiFi networks require a matching transmission medium module to be present in the network, which is usually an `Ieee80211ScalarRadioMedium`.

The operation mode (infrastructure vs ad hoc) is determined by the ingredients of the wireless interface. The `Ieee80211Interface` has the following submodules (incomplete list):

1. *management*: performs association/disassociation with access points, channel scanning, beaconing
2. *agent*: initiates actions such as channel scanning and connecting to and disconnecting from access points
3. *MAC*: transmits and receives frames according to the IEEE 802.11 medium access procedure
4. *PHY*: represents the radio

The management component has several implementations which differ in their role and level of detail:

- `Ieee80211MgmtAdhoc`: for ad hoc mode stations
- `Ieee80211MgmtSta`, `Ieee80211MgmtStaSimplified`: for infrastructure mode stations
- `Ieee80211MgmtAp`, `Ieee80211MgmtApSimplified`: for access points

The "simplified" ones assume that stations are statically associated with an access point for the entire duration of the simulation (the scan-authenticate-associate process is not simulated), so they cannot be used e.g. in experiments involving handover.

The `Ieee80211MgmtSta` does not take any action by itself, it requires an agent (`Ieee80211AgentSta` or a custom one) to initiate actions.

The following sections examine the above components.

21.2 MAC

The `Ieee80211Mac` module type represents the IEEE 802.11 MAC. The structure of the implementation closely follows the architecture described in the standard, IEEE 802.11-2012 Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.

The `Ieee80211Mac` performs transmission of frames according to the CSMA/CA protocol. Data and management frames are received from the upper layers and transmitted using the `Ieee80211Radio`.

The `Ieee80211Mac` was designed with modularity in mind to facilitate experimentation with new policies, features, and algorithms within the MAC layer. Individual components can be easily replaced with custom implementations. Policies, which are most likely to be experimented with, are extracted into their own modules.

The new model also separates the following components from the 802.11 standard into modules:

- Coordination function: `Dcf`, `Hcf`. The coordination functions control the medium access as specified by the standard.
- Channel access method as specified by the standard: `Edca`
- Channel access function: `Edcaf`, `Dcaf`. The channel access function controls channel ownership, etc.
- MAC data services: `OriginatorMacDataService`, `RecipientMacDataService`, `OriginatorQosMacDataService`, `RecipientQosMacDataService`. The MAC data services are implemented as compound modules, which combine functions related to transforming data frames (sequence numbering, duplicate removal, frame reordering, fragmentation, aggregation).
- Aggregation and deaggregation (C++ class in MAC data services): These classes implement aggregation and deaggregation according to the policy modules (see below).
- Fragmentation and defragmentation (C++ classes in MAC data services): These classes carry out the task of fragmentation and defragmentation according to the policy modules (see below).
- Block ACK agreements and frame reordering (C++ classes): These classes carry out the task according to the policy modules (see below).
- Frame exchange sequences (implemented as C++ classes in coordination functions, like `Hcf`): These classes strictly follow the standard specifying the valid frame sequences.
- TXOP procedure (`TxopProcedure`): This module implements the transmission opportunity behavior of the standard.
- Duplicate removal (C++ class in RecipientMacDataService): This class removes duplicate received frames based on sequence numbers.
- Rate selection: `QosRateSelection`. This module controls the data rate for all kinds of frames, including management and control frames.
- Rate control: `AarfRateControl`, `OnoeRateControl`. These modules determine the optimal data rate for data frames.
- Protection mechanism: `OriginatorProtectionMechanism`. This module provides channel allocation for frame exchange sequences.
- Recovery procedure: `NonQosRecoveryProcedure`, `QosRecoveryProcedure`. These modules determine what to do in case of frame exchange failure.
- Contention: `Contention`. The contention module implements contention-based channel access, using defer, backoff, etc.
- **`PendingQueue`: This module is a queue containing frames received from** higher layers, waiting for transmission unchanged.
- **`InProgressFrames`: This modules is a queue containing frames waiting** for transmission that have been processed by the MAC data service.
- TX/RX (`Tx`, `Rx`). Responsible for simple frame transmission/reception.

The MAC model has the following replaceable built-in policy submodules by default:

- ACK policy (e.g. `OriginatorAckPolicy`, `RecipientAckPolicy`): controls the type of acknowledgment used per-frame (none, normal, block-ack)
- RTS/CTS policy (e.g. `RtsPolicy`, `CtsPolicy`): determines which frames are protected by the RTS/CTS mechanism
- Originator and recipient block ACK agreement policies (e.g. `OriginatorBlockAckAgreementPolicy`, `RecipientBlockAckAgreementPolicy`): determine when and what kind of agreements are made
- MSDU aggregation policy (e.g. `BasicMsduAggregationPolicy`): controls which frames are aggregated into an A-MSDU and when
- Fragmentation policy (e.g. `BasicFragmentationPolicy`): controls when and how fragmentation occurs

21.3 Physical Layer

The physical layer modules (`Ieee80211Radio`) deal with modeling transmission and reception of frames. They model the characteristics of the radio channel and determine if a frame was received correctly (that is, it did not suffer bit errors due to low signal power or interference in the radio channel). Frames received correctly are passed up to the MAC.

On the physical layer, one can choose from several radios with different levels of detail. The various radio types (with the matching transmission medium types in parentheses) are:

- `Ieee80211ScalarRadio` (`Ieee80211ScalarRadioMedium`)
- `Ieee80211DimensionalRadio` (`Ieee80211DimensionalRadioMedium`)
- `Ieee80211UnitDiskRadio` (`UnitDiskRadioMedium`)

21.4 Management

The management layer exchanges management frames via the MAC with its peer management entities in other STAs and APs. Beacon, Probe Request/Response, Authentication, Association Request/Response, etc frames are generated and interpreted by management entities and transmitted/received via the MAC layer. During scanning, it is the management entity that periodically switches channels and collects information from received beacons and probe responses.

The management layer has several implementations which differ in their role (STA/AP/ad-hoc) and level of detail: `Ieee80211MgmtAdhoc`, `Ieee80211MgmtAp`, `Ieee80211MgmtApSimplified`, `Ieee80211MgmtSta`, `Ieee80211MgmtStaSimplified`. The ..Simplified ones differ from the others in that they do not simulate the scan-authenticate-associate process, so they cannot be used in experiments involving handover.

21.5 Agent

The agent is what instructs the management layer to perform scanning, authentication, and association. The management layer itself just carries out these commands by performing the scanning, authentication, and association procedures and reports back the results to the agent.

The agent component is currently only needed with the `Ieee80211MgmtSta` module. The management entities in other NIC variants do not have as much freedom and do not require an agent.

`Ieee80211MgmtSta` requires an `Ieee80211AgentSta` or a custom agent. By modifying or replacing the agent, one can alter the dynamic behavior of STAs in the network, for example, implementing different handover strategies.

CHAPTER

TWENTYTWO

THE 802.15.4 MODEL

22.1 Overview

IEEE 802.15.4 is a technical standard that defines the operation of low-rate wireless personal area networks (LR-WPANs). IEEE 802.15.4 was designed for data rates of 250 kbit/s or lower to achieve long battery life (months or even years) and very low complexity. The standard specifies the physical layer and media access control.

IEEE 802.15.4 is the basis for the ZigBee, ISA100.11a, WirelessHART, MiWi, SNAP, and the Thread specifications, each of which further extends the standard by developing the upper layers that are not defined in IEEE 802.15.4. Alternatively, it can be used with 6LoWPAN, the technology used to deliver IPv6 over WPANs, in order to define the upper layers. (Thread is also 6LoWPAN-based.)

A basic implementation of the IEEE 802.15.4 protocol is provided by the INET Framework.

22.2 Network Interfaces

Two network interfaces differing in the type of radio are available:

- `Ieee802154NarrowbandInterface` is for use with narrowband radios.
- `Ieee802154UwbIrInterface` is for use with the UWB-IR radio.

In order to create a wireless node with an 802.15.4 interface, use a node type that has a wireless interface and set the interface type to the appropriate type. For example, `WirelessHost` is a node type that is preconfigured to have one wireless interface, `wlan[0]`. `wlan[0]` is of parametric type, so it allows configuring of all the nodes to use 802.15.4 with the following line in the ini file:

```
**.wlan[0].typename = "Ieee802154NarrowbandInterface"
```

22.3 Physical Layer

The IEEE 802.15.4 standard defines several alternative PHYs. Various frequency bands, modulation schemes (DSSS, O-QPSK, MPSK, GFSK BPSK, etc.), and narrowband radios are deployed. The standard also includes a Direct Sequence ultra-wideband (UWB) radio and one using chirp spread spectrum (CSS).

INET provides the following radios:

- `Ieee802154NarrowbandScalarRadio` is currently a partially parameterized version of the APSK radio. Before using this radio, the radio's parameters must be checked and ensure their correspondence to the specification of the narrowband PHY being simulated.
- `Ieee802154UwbIrRadio` models the 802.14.5 UWB radio.

To match the radios, a relevant medium model should be chosen. For instance, `Ieee802154UwbIrRadioMedium` for `Ieee802154UwbIrRadio`, and `Ieee802154NarrowbandScalarRadioMedium` for `Ieee802154NarrowbandScalarRadio`.

22.4 MAC Protocol

The 802.15.4 MAC is based on collision avoidance via CSMA/CA. Other important features include real-time suitability by reservation of guaranteed time slots and integrated support for secure communications. Devices also include power management functions such as link quality and energy detection.

The `Ieee802154Mac` type in INET is currently a parameterized version of a generic CSMA/CA protocol model with ACK support.

There is also a `Ieee802154NarrowbandMac`.

CHAPTER

TWENTYTHREE

MAC PROTOCOLS FOR WIRELESS SENSOR NETWORKS

23.1 Overview

The INET Framework contains the implementation of several MAC protocols for wireless sensor networks (WSNs), including B-MAC, L-MAC, and X-MAC.

To create a wireless node with a specific MAC protocol, a node type that has a wireless interface can be used and the interface type can be set to the appropriate type. For example, `WirelessHost` is a node type which is preconfigured to have one wireless interface, `wlan[0]`. `wlan[0]` is of parametric type, so if the network is built from `WirelessHost` nodes, all of them can be configured to use, for example, B-MAC with the following line in the ini file:

```
**.wlan[0].typename = "BMacInterface"
```

23.2 B-MAC

B-MAC (Berkeley MAC) is a carrier sense media access protocol for wireless sensor networks that provides a flexible interface to obtain ultra-low power operation, effective collision avoidance, and high channel utilization. To achieve low-power operation, B-MAC employs an adaptive preamble sampling scheme to reduce duty cycle and minimize idle listening. B-MAC is designed for low-traffic, low-power communication and is one of the most widely used protocols (e.g., it is part of TinyOS).

The `BMac` module type implements the B-MAC protocol.

`BMacInterface` is a `WirelessInterface` with the MAC type set to `BMac`.

23.3 L-MAC

L-MAC (Lightweight MAC) is an energy-efficient medium access protocol designed for wireless sensor networks. Although the protocol uses TDMA to give nodes in the WSN the opportunity to communicate collision-free, the network is self-organizing in terms of time slot assignment and synchronization. The protocol reduces the number of transceiver state switches and hence the energy wasted in preamble transmissions.

The `LMac` module type implements the L-MAC protocol, based on the paper "A lightweight medium access protocol (LMAC) for wireless sensor networks" by van Hoesel and P. Havinga.

`LMacInterface` is a `WirelessInterface` with the MAC type set to `LMac`.

23.4 X-MAC

X-MAC is a low-power MAC protocol for wireless sensor networks (WSNs). In contrast to B-MAC, which employs an extended preamble and preamble sampling, X-MAC uses a shortened preamble that reduces latency at each hop and improves energy consumption while retaining the advantages of low-power listening, namely low-power communication, simplicity, and a decoupling of transmitter and receiver sleep schedules.

The XMac module type implements the X-MAC protocol, based on the paper "X-MAC: A Short Preamble MAC Protocol for Duty-Cycled Wireless Sensor Networks" by Michael Buettner, Gary V. Yee, Eric Anderson, and Richard Han.

XMacInterface is a WirelessInterface with the MAC type set to XMac.

CHAPTER

TWENTYFOUR

CLOCK MODEL

24.1 Overview

In most communication network simulations, time is simply modeled as a global quantity. All components of the network share the same time throughout the simulation, independent of their physical location or logical connection to the network.

In contrast, in time-sensitive networking, the tracking of time is an essential part, which should be explicitly simulated independently of the underlying global time. This is because the differences in the local time of the communication network components significantly affect the simulation results.

In such simulations, hardware clocks are simulated separately, and communication protocols do not rely on the global value of simulation time, which is, in fact, unknown in reality, but rather on the value of their own clocks. With hardware clocks being modeled, it is also often necessary to use various time synchronization protocols because clocks tend to drift over time and communication protocols rely on the precision of the clocks they are using.

In INET, the clock model is a completely optional feature that has no effect on the simulation performance when disabled. Even if the feature is enabled, the usage of clock modules by communication protocols and applications is still optional, and enabling the feature has a negligible performance impact when not in use. To test that the mere usage of a clock has no effect on the simulation results, INET also includes an ideal clock mechanism.

24.2 Clocks

Clocks are implemented as modules and are used by other modules via direct C++ method calls. Clock modules implement the `IClock` module interface and the corresponding `IClock` C++ interface.

The C++ interface provides an API similar to the standard OMNeT++ simulation time-based scheduling mechanism, but it relies on the underlying clock implementation for (re)scheduling events according to the clock. These events are transparently scheduled for the client module, and they will be delivered to it when the clock timers expire.

The clock API uses the clock time instead of the simulation time as arguments and return values. The interface contains functions such as `getClockTime()`, `scheduleClockEventAt()`, `scheduleClockEventAfter()`, `cancelClockEvent()`.

INET contains optional clock modules (not used by default) at the network node and the network interface levels. The following clock models are available:

- `IdealClock`: clock time is identical to the simulation time.
- `OscillatorBasedClock`: clock time is the number of oscillator ticks multiplied by the nominal tick length.
- `SettableClock`: a clock that can be set to a different clock time.

24.3 Clock Time

To avoid confusing the simulation time (which is basically unknown to communication protocols and hardware elements) with the clock time maintained by hardware clocks, INET introduces a new C++ type called the `ClockTime`.

This type is similar to the default `SimTime`, but the two types cannot be implicitly converted into each other. This approach prevents accidentally using clock time where simulation time is needed, and vice versa. Similarly to how `simtime_t` is an alias for `SimTime`, INET also introduces the `clocktime_t` alias for the `ClockTime` type.

For the explicit conversion between clock time and simulation time, one can use the `CLOCKTIME_AS_SIMTIME` and the `SIMTIME_AS_CLOCKTIME` C++ macros. Note that these macros do not change the numerical value; they simply convert between the C++ types.

When the actual clock time is used by a clock, the value may be rounded according to the clock granularity and rounding mode (e.g., `OscillatorBasedClock`). For example, when a clock with a microsecond granularity is instructed to wait for 100 nanoseconds, while its oscillator is right in the middle of its ticking period, it may actually wait for the next tick to happen to start the timer and wait another tick to account for the requested wait time interval.

24.4 Oscillators

The clock interface is quite general in the sense that it allows for many different ways of implementation. Nevertheless, the most common way is to use an oscillator-based clock model.

An oscillator efficiently models the periodic generation of ticks that are usually counted by a clock module. The tick period is not necessarily constant; it can change over time. Oscillators implement the `IOscillator` module interface and the corresponding `IOscillator` C++ interface.

The following oscillator models are available:

- `IdealOscillator`: ticks are generated periodically with a constant length (mostly useful for testing).
- `ConstantDriftOscillator`: tick length changes proportionally to the elapsed simulation time (clock drift).
- `RandomDriftOscillator`: updates clock drift with a random walk process.

24.5 Clock Users

The easiest way to use a clock in applications and communication protocols is to add a *clockModule* parameter that specifies where the clock module can be found. Then the C++ user module should simply be derived from either `ClockUserModuleBase` or the parameterizable `ClockUserModuleMixin` base classes. The clock can be used via the inherited clock-related methods or through the methods of the `IClock` C++ interface on the inherited clock field.

24.6 Clock Events

The clock model requires the use of a specific C++ class called `ClockEvent` to schedule clock timers. It is also allowed to derive new C++ classes from `ClockEvent` if necessary. In any case, clock events must be scheduled and canceled via the `IClock` C++ interface to operate correctly.

24.7 Controlling Clocks According to a Scenario

To support the simulation of specific scenarios, where the clock time or the oscillator drift must be changed according to a predefined script, INET provides clocks and oscillators that implement the interface required by the `ScenarioManager` module. This allows the user to update the clock and oscillator states from the `ScenarioManager` XML script and also to mix these operations with many other supported operations.

For example, the `SettableClock` model supports setting the clock time and optionally resetting the oscillator at a specific moment of simulation time as follows:

```
<set-clock at="10 s" module="server.clock" time="1.2 s" reset-oscillator="true"/>
```

The above example means that the clock time of the server node's clock will be set to 1.2 seconds when the simulation time reaches 10 seconds, and the clock's oscillator will restart its duty cycle.

For another example, the `ConstantDriftOscillator` supports changing the state of the oscillator with the following command:

```
<set-oscillator at="10 us" module="server.clock.oscillator" drift-rate="42 ppm" tick-
↪offset="1 us"/>
```

This example simultaneously changes the drift rate and the tick offset of the oscillator in the server node's clock.

CHAPTER

TWENTYFIVE

THE PHYSICAL LAYER

25.1 Overview

Wireless network interfaces contain a radio model component, which is responsible for modeling the physical layer (PHY).[1] The radio model describes the physical device that is capable of transmitting and receiving signals on the medium.

Conceptually, a radio model relies on several sub-models:

- antenna model
- transmitter model
- receiver model
- error model (as part of the receiver model)
- energy consumption model

The antenna model is shared between the transmitter model and the receiver model. The separation of the transmitter model and the receiver model allows asymmetric configurations. The energy consumer model is optional, and it is only used when the simulation of energy consumption is necessary.

25.2 Generic Radio

In INET, radio models implement the `IRadio` module interface. A generic, often used implementation of `IRadio` is the `Radio` NED type. `Radio` is an active compound module, that is, it has an associated C++ class that encapsulates the computations.

`Radio` contains its antenna, transmitter, receiver, and energy consumer models as submodules with parametric types:

```
antenna: <default("IsotropicAntenna")> like IAntenna;
transmitter: <> like ITransmitter;
receiver: <> like IReceiver;
energyConsumer: <default("")> like IEnergyConsumer
    if typename != "";
```

The following sections describe the parts of the radio model.

25.3 Components of a Radio

25.3.1 Antenna Models

The antenna model describes the effects of the physical device that converts electric signals into radio waves, and vice versa. This model captures the antenna characteristics that heavily affect the quality of the communication

[1] Wired network interfaces could similarly contain an explicit PHY model. The reason they do not is that wired links normally have very low error rates and simple observable behavior, and there is usually not much to be gained from modeling the physical layer in detail.

channel. For example, various antenna shapes, antenna size and geometry, antenna arrays, and antenna orientation cause different directional or frequency selectivity.

The antenna model provides a position and an orientation using a mobility model that defaults to the mobility of the node. The main purpose of this model is to compute the antenna gain based on the specific antenna characteristics and the direction of the signal. The signal direction is computed by the medium from the position and the orientation of the transmitter and the receiver. The following list provides some examples:

- `IsotropicAntenna`: antenna gain is exactly 1 in any direction
- `ConstantGainAntenna`: antenna gain is a constant determined by a parameter
- `DipoleAntenna`: antenna gain depends on the direction according to the dipole antenna characteristics
- `InterpolatingAntenna`: antenna gain is computed by linear interpolation according to a table indexed by the direction angles

25.3.2 Transmitter Models

The transmitter model describes the physical process that converts packets into electric signals. In other words, this model converts an L2 frame into a signal that is transmitted on the medium. The conversion process and the representation of the signal depend on the level of detail and the physical characteristics of the implemented protocol.

There are two main levels of detail (or modeling depths):

- In the *flat model*, the transmitter model skips the symbol domain and the sample domain representations, and it directly creates the analog domain representation. The bit domain representation is reduced to the bit length of the packet, and the actual bits are ignored.
- In the *layered model*, the conversion process involves various processing steps such as packet serialization, forward error correction encoding, scrambling, interleaving, and modulation. This transmitter model requires significantly more computation, but it produces accurate bit domain, symbol domain, and sample domain representations.

Some of the transmitter types available in INET:

- `GenericTransmitter`
- `ApskTransmitter`
- `ApskLayeredTransmitter`
- `Ieee80211Transmitter`

25.3.3 Receiver Models

The receiver model describes the physical process that converts electric signals into packets. In other words, this model converts a reception, along with an interference computed by the medium model, into a MAC packet and a reception indication.

For a packet to be received successfully, reception must be *possible* (based on reception power, bandwidth, modulation scheme, and other characteristics), it must be *attempted* (i.e. the receiver must synchronize itself on the preamble and start receiving), and it must be *successful* (as determined by the error model and the simulated part of the signal decoding).

In the *flat model*, the receiver model skips the sample domain, the symbol domain, and the bit domain representations, and it directly creates the packet domain representation by copying the packet from the transmission. It uses the error model to decide whether the reception is successful.

In the *layered model*, the conversion process involves various processing steps such as demodulation, descrambling, deinterleaving, forward error correction decoding, and deserialization. This reception model requires much more computation than the flat model, but it produces accurate sample domain, symbol domain, and bit domain representations.

Some of the receiver types available in INET:

- `GenericReceiver`
- `ApskReceiver`
- `ApskLayeredReceiver`
- `Ieee80211Receiver`

25.3.4 Error Models

Determining reception errors is a crucial part of the reception process. There are often several different statistical error models in the literature even for a particular physical layer. To support this diversity, the error model is a separate replaceable component of the receiver.

The error model describes how the signal-to-noise ratio affects the amount of errors at the receiver. The main purpose of this model is to determine whether the received packet has errors or not. It also computes various physical layer indications for higher layers, such as packet error rate, bit error rate, and symbol error rate. For the layered reception model, it needs to compute the erroneous bits, symbols, or samples depending on the lowest simulated physical domain where the real decoding starts. The error model is optional (if omitted, all receptions are considered successful).

The following list provides some examples:

- `StochasticErrorModel`: simplistic error model with constant symbol/bit/packet error rates as parameters; suitable for testing.
- `ApskErrorModel`
- `Ieee80211NistErrorModel`, `Ieee80211YansErrorModel`, `Ieee80211BerTableErrorModel`: various error models for IEEE 802.11 network interfaces.

25.3.5 Power Consumption Models

A substantial part of the energy consumption of communication devices comes from transmitting and receiving signals. The energy consumer model describes how the radio consumes energy depending on its activity. This model is optional (if omitted, energy consumption is ignored).

The following list provides some examples:

- `StateBasedEpEnergyConsumer`: power consumption is determined by the radio state (a combination of radio mode, transmitter state, and receiver state), and specified in parameters like `receiverIdlePowerConsumption` and `receiverReceivingDataPowerConsumption`, in watts.
- `StateBasedCcEnergyConsumer`: similar to the previous one, but consumption is given in amperes.

25.4 Layered Radio Models

In layered radio models, the transmitter and receiver models are split into several stages to allow more fine-grained modeling.

For transmission, processing steps such as packet serialization, forward error correction (FEC) encoding, scrambling, interleaving, and modulation are explicitly modeled. Reception involves the inverse operations: demodulation, descrambling, deinterleaving, FEC decoding, and deserialization.

In layered radio models, these processing steps are encapsulated in four stages, represented as four submodules in both the transmitter and receiver model:

1. *Encoding and Decoding* describe how the packet domain signal representation is converted into the bit domain, and vice versa.
2. *Modulation and Demodulation* describe how the bit domain signal representation is converted into the symbol domain, and vice versa.
3. *Pulse Shaping and Pulse Filtering* describe how the symbol domain signal representation is converted into the sample domain, and vice versa.

4. *Digital Analog and Analog Digital Conversion* describe how the sample domain signal representation is converted into the analog domain, and vice versa.

In layered radio transmitters and receivers such as `ApskLayeredTransmitter` and `ApskLayeredReceiver`, these submodules have parametric types to make them replaceable. This provides immense freedom for experimentation.

25.5 Notable Radio Models

The `Radio` module has several specialized versions derived from it, where certain submodule types and parameters are set to fixed values. This section describes some of the frequently used ones.

The radio can be replaced in wireless network interfaces by setting the `typename` parameter of the radio submodule, like in the following ini file fragment.

```
**.wlan[*].radio.typename = "GenericUnitDiskRadio"
```

However, be aware that not all MAC protocols can be used with all radio models, and that some radio models require a matching transmission medium module.

25.5.1 GenericUnitDiskRadio

`GenericUnitDiskRadio` provides a very simple but fast and predictable physical layer model. It is the implementation (with some extensions) of the *Unit Disk Graph* model, which is widely used for the study of wireless ad-hoc networks. `GenericUnitDiskRadio` is applicable if network nodes need to have a finite communication range, but physical effects of signal propagation are to be ignored.

`GenericUnitDiskRadio` allows three radii to be given as parameters, instead of the usual one: communication range, interference range, and detection range. One can also turn off interference modeling (meaning that signals colliding at a receiver will all be received correctly), which is sometimes a useful abstraction.

`GenericUnitDiskRadio` needs to be used together with a special physical medium model, `UnitDiskRadioMedium`.

The following ini file fragment shows an example configuration.

```
*.radioMedium.typename = "UnitDiskRadioMedium"
*.host[*].wlan[*].radio.typename = "GenericUnitDiskRadio"
*.host[*].wlan[*].radio.transmitter.bitrate = 2Mbps
*.host[*].wlan[*].radio.transmitter.preambleDuration = 0s
*.host[*].wlan[*].radio.transmitter.headerLength = 96b
*.host[*].wlan[*].radio.transmitter.analogModel.communicationRange = 100m
*.host[*].wlan[*].radio.transmitter.analogModel.interferenceRange = 0m
*.host[*].wlan[*].radio.transmitter.analogModel.detectionRange = 0m
*.host[*].wlan[*].radio.receiver.ignoreInterference = true
```

As a side note, if modeling full connectivity and ignoring interference is required, then `ShortcutInterface` provides an even simpler and faster alternative.

25.5.2 APSK Radio

APSK radio models provide a hypothetical radio that simulates one of the well-known APSK, PSK, and QAM modulations. (APSK stands for Amplitude and Phase-Shift Keying.)

APSK radio has scalar/dimensional and flat/layered variants. The flat variants, `ApskScalarRadio` and `ApskDimensionalRadio`, model frame transmissions in the selected modulation scheme but without utilizing other techniques such as forward error correction (FEC), interleaving, spreading, etc. These radios require matching medium models, `ScalarRadioMedium` and `DimensionalRadioMedium`.

The layered version, `ApskLayeredRadio`, can not only model the processing steps missing from their simpler counterparts, but they also feature a configurable level of detail: the transmitter and receiver modules have

`levelOfDetail` parameters that control which domains are actually simulated. These radio models must be used in conjunction with `DimensionalRadioMedium`.

CHAPTER

TWENTYSIX

THE TRANSMISSION MEDIUM

26.1 Overview

For wireless communication, an additional module is required to model the shared physical medium where the communication takes place. This module keeps track of transceivers, noise sources, ongoing transmissions, background noise, and other ongoing noises.

It relies on several models:

1. Signal propagation model
2. Path loss model
3. Obstacle loss model
4. Background noise model
5. Medium analog model

With the help of the above models, the medium module computes when, where, and how signals arrive at receivers, including the set of interfering signals and noises. In addition, the medium module also contains various mechanisms and ways to improve the scalability of wireless network simulations.

26.2 RadioMedium

The standard transmission medium model in the INET Framework is `RadioMedium`. `RadioMedium` is an OMNeT++ compound module with several replaceable submodules. Submodules for each of the above models (signal propagation, path loss, etc.) and various caches for efficiency are included.

Note that `RadioMedium` is an active compound module, that is, it has an associated C++ class that encapsulates the computations.

`RadioMedium` contains its components as submodules with parametric types:

```
propagation: <default("ConstantSpeedPropagation")> like IPropagation;
analogModel: <default("ScalarMediumAnalogModel")> like IMediumAnalogModel;
backgroundNoise: <default("IsotropicScalarBackgroundNoise")> like␣
↪IRadioBackgroundNoise
    if typename != "";
pathLoss: <default("FreeSpacePathLoss")> like IPathLoss;
obstacleLoss: <default("")> like IObstacleLoss
    if typename != "";
mediumLimitCache: <default("MediumLimitCache")> like IMediumLimitCache;
communicationCache: <default("VectorCommunicationCache")> like ICommunicationCache;
neighborCache: <default("")> like INeighborCache
    if typename != "";
```

There are many preconfigured versions of `RadioMedium`:

- For use with `GenericUnitDiskRadio`: `UnitDiskRadioMedium`

- For APSK radios: `ScalarRadioMedium`, `DimensionalRadioMedium`
- For IEEE 802.11: `Ieee80211ScalarRadioMedium`, `Ieee80211DimensionalRadioMedium`.
- For IEEE 802.15.4: `Ieee802154UwbIrRadioMedium`, `Ieee802154NarrowbandScalarRadioMedium`.

The following sections describe the parts of the medium model.

26.3 Propagation Models

When a signal transmitter starts transmission, the signal's start propagates through the transmission medium. When the transmitter ends transmission, the signal's end propagates similarly. The propagation model describes how a signal moves through space over time. Its main purpose is to compute the arrival space-time coordinates at receivers. INET implements two built-in models:

- `ConstantTimePropagation`: A simplistic model where the propagation time is independent of the travelled distance. The propagation time is simply determined by a module parameter.
- `ConstantSpeedPropagation`: A more realistic model where the propagation time is proportional to the travelled distance. The propagation time is independent of the transmitter and receiver movement during both signal transmission and propagation. The propagation speed is determined by a module parameter.

The default propagation model is configured as follows:

```
*.radioMedium.propagation.typename = "ConstantSpeedPropagation" # module type
*.radioMedium.propagation.propagationSpeed = 299792458 mps # speed of light
```

A more accurate model could take into consideration the transmitter and receiver movement. This effect becomes especially important for acoustic communication, because the propagation speed of the signal is much more comparable to the speed of the transceivers.

26.4 Path Loss Models

As a signal propagates through space, its power density decreases. This is called path loss and it is the combination of many effects such as free-space loss, refraction, diffraction, reflection, and absorption. There are several different path loss models in the literature, which differ in their parameterization and application area.

In INET, a path loss model is an OMNeT++ simple module implementing a specific path loss algorithm. Its main purpose is to compute the power loss for a given signal, but it is also capable of estimating the range for a given loss. The latter is useful, for example, to allow visualizing communication range. INET contains a number of built-in path loss algorithms, each comes with its own set of parameters:

- `FreeSpacePathLoss` models line of sight path loss for air or vacuum.
- `BreakpointPathLoss` refines it using dual slope model with two separate path loss exponents.
- `LogNormalShadowing` models path loss for a wide range of environments (e.g., urban areas, and buildings)
- `TwoRayGroundReflection` models interference between line of sight and single ground reflection.
- `TwoRayInterference` refines the above for inter-vehicle communication.
- `RicianFading` is a stochastic model for the anomaly caused by partial cancellation of a signal by itself.
- `RayleighFading` is a stochastic model for heavily built-up urban environments when there is no dominant propagation along the line of sight.
- `NakagamiFading` further refines the above two models for cellular systems.

The following example replaces the default free-space path loss model with log normal shadowing:

```
*.radioMedium.pathLoss.typename = "LogNormalShadowing" # module type
*.radioMedium.pathLoss.sigma = 1.1 # override default value of 1
```

26.5 Obstacle Loss Models

When the signal propagates through space, it also passes through physical objects present in that space. As the signal penetrates physical objects, its power decreases when it reflects from surfaces, and also when it is absorbed by their material. There are various ways to model this effect, which differ in the trade-off between accuracy and performance.

In INET, an obstacle loss model is an OMNeT++ simple module. Its main purpose is to compute the power loss based on the traveled path and the signal frequency. The obstacle loss models most often use the physical environment model to determine the set of penetrated physical objects. INET contains a few built-in obstacle loss models:

- `IdealObstacleLoss` model determines total or no power loss at all by checking if there is any obstructing physical object along the straight propagation path.
- `DielectricObstacleLoss` computes the power loss based on the accurate dielectric and reflection loss along the straight path considering the shape, position, orientation, and material of obstructing physical objects.

By default, the medium module doesn't contain any obstacle loss model, but configuring one is very simple:

```
*.radioMedium.obstacleLoss.typename = "DielectricObstacleLoss" # module type
```

Statistical obstacle loss models are also possible but currently not provided.

26.6 Background Noise Models

Thermal noise, cosmic background noise, and other random fluctuations of the electromagnetic field affect the quality of the communication channel. This kind of noise doesn't come from a particular source, so it doesn't make sense to model its propagation through space. The background noise model describes instead how it changes over space and time.

In INET, a background noise model is an OMNeT++ simple module. Its main purpose is to compute the analog representation of the background noise for a given space-time interval. For example, `IsotropicScalarBackgroundNoise` computes a background noise that is independent of space-time coordinates, and its scalar power is determined by a module parameter.

The simplest background noise model can be configured as follows:

```
*.radioMedium.backgroundNoise.typename = "IsotropicScalarBackgroundNoise" # type
*.radioMedium.backgroundNoise.power = -110 dBm # isotropic scalar noise power
```

26.7 Medium Analog Models

The analog signal is a complex physical phenomenon that can be modeled in many different ways. Choosing the right analog domain signal representation is the most important factor in the trade-off between accuracy and performance. The analog model of the transmission medium determines how signals are represented while being transmitted, propagated, and received.

In INET, a medium analog model is an OMNeT++ simple module. Its main purpose is to compute the received signal from the transmitted signal. The analog model combines the effect of the antenna, path loss, and obstacle loss models. Transceivers must be configured to transmit and receive signals according to the representation used by the analog model.

The most commonly used medium analog model, which uses a scalar signal power representation over a frequency and time interval, can be configured as follows:

```
*.radioMedium.analogModel.typename = "ScalarMediumAnalogModel" # module type
```

26.8 Neighbor Cache

Transceivers are considered neighbors if successful communication is possible between them. For wired communication, it is easy to determine which transceivers are neighbors because they are connected by wires. In contrast, in wireless communication, determining which transceivers are neighbors isn't obvious at all.

In INET, a neighbor cache is an OMNeT++ simple module that provides an efficient way of keeping track of the neighbor relationship between transceivers. Its main purpose is to compute the set of affected receivers for a given transmission. All built-in models in INET provide a conservative approximation only because they update their state periodically:

- `NeighborListNeighborCache` takes a range as a parameter, and for each transceiver, it maintains the list of receivers within range (neighbor list).
- `GridNeighborCache` organizes transceivers in a 3D grid with a constant cell size.
- `QuadTreeNeighborCache` organizes transceivers in a 2D quad tree (ignoring the Z axis) with a constant node size.

The following example sets `QuadTreeNeighborCache` as the neighbor cache:

```
**.radioMedium.neighborCache.typename = "QuadTreeNeighborCache" # module type
**.radioMedium.neighborCache.maxNumOfPointsPerQuadrant = 4 # affects tree depth
```

How should one chose the neighbor cache for a given simulation? As the sole purpose of the neighbor cache is to speed up the simulation, one should choose the one that leads to the best performance for that particular network. Which one performs the best is best determined by experimentation, as it depends on many factors: the number of nodes, their spatial distribution, their speed and movement pattern, their communication pattern, and so on. Note that not only the choice of the neighbor cache but also its parameterization can affect performance.

26.9 Medium Limit Cache

The medium limit cache (and its default implementation `MediumLimitCache`) keeps track of certain thresholds and minimum/maximum values of quantities related to layer 1 modeling. Some of these limits can be gathered from other modules in the network, but still, all of them can be explicitly specified by the user. The quantities include:

- Maximum speed (can be gathered from mobility models)
- Maximum transmission power
- Minimum interference power and reception power
- Maximum antenna gain (can be computed from antenna models)
- Minimum time interval to consider two overlapping signals interfering
- Maximum duration of a transmission
- Maximum communication range and interference range (can be computed from transmitter and receiver models)

These limits allow the transmission medium model to make assumptions about the locations of nodes (i.e. the maximum distance they can move during some interval), about the possibility of interference, and about the possibility of a signal being receivable.

26.10 Communication Cache

The communication cache is used to cache various intermediate computation results related to the communication on the medium. The main motivation to have multiple implementations is that different implementations may be the most efficient in different simulations. Also, a conservative (simple but robust) implementation may be used for validating new (more efficient but also more complex) implementations.

Implementations include:

- `ReferenceCommunicationCache`
- `MapCommunicationCache`
- `VectorCommunicationCache`

26.11 Improving Scalability

The simulation of wireless networks is inherently less scalable than that of wired networks. In wired networks, a transmission only affects the host's neighbors on the link, which is usually 1 in modern networks that are dominated by point-to-point links. The wireless medium, however, is a broadcast medium. Any transmission is "heard" by all nodes within the interference range, not only the intended recipients. The signal may be receivable by them (and must be indeed received before the destination address field in it can be examined) or may interfere with the reception of other transmissions. Whichever the case, the transmission must be evaluated or processed by a much larger number of nodes than in the wired case. This makes the computational complexity at least $O(n^2)$ (n being the number of nodes.) Other effects may further increase the exponent.

The medium module provides a set of parameters that can be used to alleviate the scalability issue. These *filter* parameters that can be used to reduce the amount of processing at nodes that are not the intended recipients of the frame, increasing simulation performance.

There are several filters that can be enabled/disabled individually:

- *Range filter*. When this filter is active, the medium module does not send signals to a radio if it is outside the interference range (or communication range, this option can also be selected.)
- *Radio mode filter*. When this filter is active, the medium module does not send signals to a radio if it is neither in *receiver* nor in *transceiver* mode.
- *Listening filter*. When this filter is active, the medium module does not send signals to a radio if it listens on the channel in incompatible mode (e.g., different carrier frequency and bandwidth, or different modulation)
- *MAC address filter*. When this filter is active, the radio medium does not send signals to a radio if the destination MAC address does not match

The corresponding module parameters are called `rangeFilter`, `radioModeFilter`, `listeningFilter`, and `macAddressFilter`. By default, all filters are turned off.

CHAPTER

TWENTYSEVEN

THE PHYSICAL ENVIRONMENT

27.1 Overview

Wireless networks are heavily affected by the physical environment, and the requirements for today's ubiquitous wireless communication devices are increasingly demanding. Densely populated urban areas are served by cellular networks, large buildings with several offices need to be covered by wireless LANs, low-power wireless sensors must tolerate noisy industrial environments, and batteries need to remain operational under various external conditions, among other things.

The propagation of radio signals, the movement of communicating agents, battery exhaustion, etc., depend on the surrounding physical environment. For example, objects can absorb or let through signals, surfaces can refract or reflect signals, and battery capacity might depend on external temperature. These effects cannot be ignored in high-fidelity simulations.

To assist in the modeling process, the physical environment model in the INET Framework is separated from the rest of the simulation model. The main goal of the physical environment model is to describe buildings, walls, vegetation, terrain, weather, and other physical objects and conditions that might have effects on radio signal propagation, movement, batteries, etc. This separation allows the model to be reusable by all other simulation models that depend on these circumstances.

The following sections provide a brief overview of the physical environment model.

27.2 PhysicalEnvironment

In INET, the physical environment is modeled by the `PhysicalEnvironment` compound module. This module normally has one instance in the network and acts as a database that other parts of the simulation can query at runtime. It contains the following information:

- geometry and properties of *physical objects* (usually referred to as "obstacles" in wireless simulations)
- a *ground model*
- other physical properties of the environment, like its bounds in space

`PhysicalEnvironment` is an active compound module, that is, it has an associated C++ class that contains the data structures and implements an API that allows other modules to query the data.

Part of `PhysicalEnvironment`'s functionality is implemented in submodules for easy replacement. They are currently the ground model and an object cache (for efficient queries):

```
ground: <default("")> like IGround if typename != "";
objectCache: <default("")> like IObjectCache if typename != "";
```

27.3 Physical Objects

The most important aspect of the physical environment is the objects present within it. For example, simulating an indoor Wifi scenario may require modeling walls, floors, ceilings, doors, windows, furniture, and similar objects because they all affect signal propagation (obstacle modeling).

Objects are located in space and have shapes and materials. The physical environment model supports basic shapes and homogeneous materials, which is a simplified description but still allows for a reasonable approximation of reality. Physical objects in INET have the following properties:

- *shape* describes the object in 3D independent of its position and orientation.
- *position* determines where the object is located in 3D space.
- *orientation* determines how the object is rotated relative to its default orientation.
- *material* describes material-specific physical properties.
- *graphical properties* provide parameters for better visualization.

Graphical properties include:

- *line width*: affects surface outline
- *line color*: affects surface outline
- *fill color*: affects surface fill
- *opacity*: affects surface outline and fill
- *tags*: allows filtering objects on the graphical user interface

Physical objects in INET are stationary; they cannot change their position or orientation over time. Since the shape of the physical objects might be quite diverse, the model is designed to be extensible with new shapes. INET provides the following shapes:

- *sphere* shapes are specified by a radius
- *cuboid* shapes are specified by a length, width, and height
- *prism* shapes are specified by a 2D polygon base and a height
- *polyhedron* shapes are specified by the convex hull of a set of 3D vertices

The following example shows how to define various physical objects using the XML syntax supported by the physical environment:

```
<environment>
  <!-- shapes and materials -->
  <shape id="1" type="sphere" radius="10"/>
  <shape id="2" type="cuboid" size="20 30 40"/>
  <shape id="3" type="prism" height="10" points="0 0 10 0 10 10 0 10"/>
  <shape id="4" type="polyhedron" points="0 0 0 10 0 0 10 10 0 0 10 0 ..."/>
  <material id="1" resistivity="10" relativePermittivity="4.5"/>
  <!-- an object that uses a previously defined shape and material -->
  <object position="min 10 20 0" orientation="45 0 0" shape="1" material="1"/>
  <!-- an object defined with an in-line shape -->
  <object position="min 10 20 0" orientation="45 -30 0" shape="cuboid 20 30 40"
            material="concrete" line-color="0 0 0" fill-color="112 128 144"/>
</environment>
```

To load the above XML file, the following configuration could be used:

```
*.physicalEnvironment.config = xmldoc("objects.xml") # load physical objects
```

27.4 Ground Models

In inter-vehicle simulations, the terrain has profound effects on signal propagation. For example, vehicles on the opposite sides of a mountain cannot directly communicate with each other.

A ground model describes the 3D surface of the terrain. Its main purpose is to compute a position on the surface underneath a particular position.

INET contains the following built-in ground models implemented as OMNeT++ simple modules:

- `FlatGround` is a trivial model that provides a flat surface parallel to the XY plane at a certain height.
- `OsgEarthGround` is a more realistic model (based on) that provides a terrain surface.

27.5 Geographic Coordinate System Models

In order to run high-fidelity simulations, it is often required to embed the communication network into a real-world map. With the new OMNeT++ 5 version, INET already provides support for 3D maps using for visualization and as the map provider.

However, INET carries out all geometric computation internally (including signal propagation and path loss) in a 3D Euclidean coordinate system. The discrepancy between the internal scene coordinate system and the usual geographic coordinate systems must be resolved.

A geographic coordinate system model maps scene coordinates to geographic coordinates, and vice versa. Such a model allows positioning physical objects and describing network node mobility using geographical coordinates (e.g., longitude, latitude, altitude).

In INET, a geographic coordinate system model is implemented as an OMNeT++ simple module:

- `SimpleGeographicCoordinateSystem` provides a trivial linear approximation without any external dependency.
- `OsgGeographicCoordinateSystem` provides an accurate mapping using the external library.
- `Wgs84EcefGeographicCoordinateSystem` maps geographic coordinates to a geocentric (Earth-Centered, Earth-Fixed) scene based on the WGS84 ellipsoid, without any external dependency. The scene origin is the center of the Earth, the Z axis points to the geographic North pole, and the X axis lies in the equatorial plane at the prime meridian. It is the coordinate system used for satellite scenarios, where node positions span the whole globe.
- `Wgs84AnchoredGeographicCoordinateSystem` maps geographic coordinates to a *local* tangent-plane scene pinned to an anchor point (`sceneLatitude`, `sceneLongitude`, `sceneAltitude`) and an optional orientation (`sceneHeading`, `sceneElevation`, `sceneBank`), using the WGS84 ellipsoid and no external dependency. Scene coordinates stay small and centered on the anchor; with the default `sceneHeading` of 90 degrees the X axis points east, the Y axis points south, and the Z axis points up. This is convenient for local scenarios that may be placed anywhere on the globe.

In order to use geographic coordinates in a simulation, a geographic coordinate system module must be included in the network. The desired physical environment module and mobility modules must be configured (using module path parameters) to use the geographic coordinate system module. The following example also shows how the geographic coordinate system module can be configured to place the scene at a particular geographic location and orientation.

```
*.physicalEnvironment.coordinateSystemModule = "coordinateSystem" # reference
*.*.mobility.coordinateSystemModule = "coordinateSystem" # reference
*.coordinateSystem.sceneLongitude = -71.070421deg # scene origin
*.coordinateSystem.sceneLatitude = 42.357824deg # scene origin
*.coordinateSystem.sceneHeading = 68.3deg # scene orientation
```

27.6 Object Cache

If a simulation contains a large number of physical objects, then signal propagation may become computationally very expensive. The reason is that the transmission medium model must check each line of sight path between all transmitter and receiver pairs against all physical objects.

An object cache organizes physical objects into a data structure that provides efficient geometric queries. Its main purpose is to iterate all physical objects penetrated by a 3D line segment.

In INET, an object cache model is implemented as an OMNeT++ simple module:

- `GridObjectCache` organizes objects into a fixed cell size 3D spatial grid.
- `BvhObjectCache` organizes objects into a tree data structure based on recursive 3D volume division.

CHAPTER

TWENTYEIGHT

NODE MOBILITY

28.1 Overview

In order to simulate ad-hoc wireless networks, it is important to model the motion of mobile network nodes. Received signal strength, signal interference, and channel occupancy depend on the distances between nodes. The selected mobility models can significantly influence the results of the simulation (e.g. via packet loss rates).

A mobility model describes position and orientation over time in a 3D Euclidean coordinate system. Its main purpose is to provide position, velocity and acceleration, and also angular position, angular velocity, and angular acceleration data as three-dimensional quantities at the current simulation time.

In INET, a mobility model is most often an OMNeT++ simple module implementing the motion as a C++ algorithm. Although most models have a few common parameters (e.g. for initial positioning), they always come with their own set of parameters. Some models support geographic positioning to ease the configuration of map-based scenarios.

Mobility models can be *single* or *group* mobility models. Single mobility models describe the motion of entities independent of each other. Group mobility models provide such a motion where group members are dependent on each other.

Mobility models can also be categorized as *trace-based*, *deterministic*, *stochastic*, and *combining* models.

28.1.1 Using Mobility Models

In order for a mobility model to actually have an effect on the motion of a network node, the mobility model needs to be included as a submodule in the compound module of the network node. By default, a transceiver antenna within a network node uses the same mobility model as the node itself, but that is completely optional. For example, it is possible to model a vehicle facing forward while moving on a road that contains multiple transceiver antennas at different relative locations with different orientations.

28.1.2 The Scene

Many mobility models allow the user to define a cubic volume that the node cannot leave. The volume is configured by setting the `constraintAreaMinX`, `constraintAreaMinY`, `constraintAreaMinZ`, `constraintAreaMaxX`, `constraintAreaMaxY`, and `constraintAreaMaxZ` parameters.

If the `initFromDisplayString` parameter is set, the initial position is taken from the display string. Otherwise, the position can be given in the `initialX`, `initialY`, and `initialZ` parameters. If neither of these parameters is given, a random initial position is chosen within the constraint area.

When the node reaches the boundary of the constraint area, the mobility component has to prevent the node from exiting. Many mobility models offer the following policies:

- reflection off the wall
- reappearance at the opposite edge (torus area)
- placement at a randomly chosen position in the area
- stopping the simulation with an error

28.2 Built-In Mobility Models

28.2.1 List of Mobility Models

The following (possibly incomplete) list contains the mobility models available in INET. Nearly all of these models are single mobility models; group mobility can be implemented, for example, by combining other mobility models.

Stationary

Stationary models only define position (and orientation), but no motion.

- `StationaryMobility` provides deterministic and random positioning.
- `StaticGridMobility` places several mobility models in a rectangular grid.
- `StaticLinearMobility` places several mobility models evenly spaced along a line.
- `StaticConcentricMobility` places several models in a set of concentric circles.

Deterministic

Deterministic mobility models use non-random mathematical models to describe motion.

- `LinearMobility` moves linearly with a constant speed or constant acceleration.
- `CircleMobility` moves around a circle parallel to the XY plane with constant speed.
- `RectangleMobility` moves around a rectangular area parallel to the XY plane with constant speed.
- `TractorMobility` moves similarly to a tractor on a field with a number of rows.
- `VehicleMobility` moves similarly to a vehicle along a path, especially turning around corners.
- `TurtleMobility` moves according to an XML script written in a simple yet expressive LOGO-like programming language.
- `FacingMobility` orients towards the position of another mobility model.

Trace-Based

Trace-based mobility models replay recorded motion as observed in real life.

- `BonnMotionMobility` replays trace files from the BonnMotion scenario generator.
- `Ns2MotionMobility` replays files from the CMU's scenario generator used in ns2.
- `AnsimMobility` replays XML trace files from the ANSim (Ad-Hoc Network Simulation) tool.

Stochastic

Stochastic or random mobility models use mathematical models involving random numbers.

- `RandomWaypointMobility` moves to random destinations with random speed.
- `GaussMarkovMobility` uses one parameter to vary the degree of randomness from linear to Brown motion.
- `MassMobility` moves similarly to a mass with inertia and momentum.
- `ChiangMobility` uses a probabilistic transition matrix to change the motion state.

Combining

Combining mobility models are not mobility models per se, but instead, they allow for more complex motions to be formed from simpler ones via superposition and other ways.

- `SuperpositioningMobility` model combines several other mobility models by summing them up. It allows creating group mobility by sharing a mobility model in each group member, separating initial positioning from positioning during the simulation, and separating positioning from orientation.

- `AttachedMobility` models mobility that is attached to another one at a given offset. Position, velocity, and acceleration are all affected by the respective quantities and also the orientation of the referenced mobility.

Satellite and Geographic

These models position and move nodes using geographic coordinates (latitude, longitude, altitude). They rely on a geographic coordinate system module in the network, referenced through their `coordinateSystemModule` parameter. For global, Earth-scale scenarios this is typically a `Wgs84EcefGeographicCoordinateSystem`, whose scene frame is the geocentric WGS84/ECEF frame. Both models default to showing the current geographic position (latitude, longitude, altitude) above the node through the `{geo_position}` display-string directive.

- `SatelliteMobility` propagates a satellite orbit from a TLE (two-line element) set using an embedded SGP4 model, and reports the resulting sub-satellite position.
- `GnssTrackMobility` follows a recorded GNSS (WGS84) track read from a CSV, GPX, or GeoJSON source, mapping the geographic positions into the scene. It is useful for replaying real car, aircraft, or UAV routes.

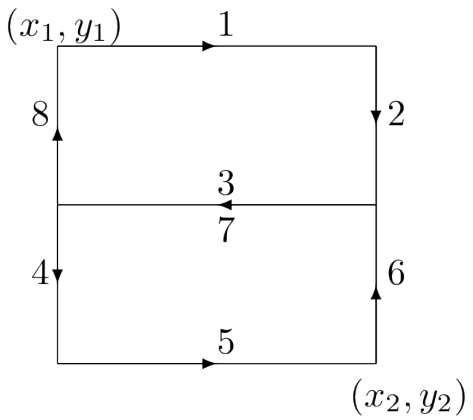
Stationary geographic nodes, such as ground stations, do not need a dedicated model: the ordinary `StationaryMobility` accepts geographic positioning through its `coordinateSystemModule` together with the `initialLatitude`, `initialLongitude`, and `initialAltitude` parameters.

To place a *local* scene at a geographic anchor (so a terrestrial scenario, e.g. a car or a UAV moving over a small area, can sit anywhere on the globe while its mobility model keeps using small local scene coordinates), use a `Wgs84AnchoredGeographicCoordinateSystem` as the coordinate system.

28.2.2 More Information on Some Mobility Models

TractorMobility

Moves a tractor through a field with a certain number of rows. The following figure illustrates the movement of the tractor when the `rowCount` parameter is 2. The trajectory follows the segments in the `1,2,3,4,5,6,7,8,1,2,3...` order. The area is configured by the `x1`, `y1`, `x2`, `y2` parameters.



RandomWaypointMobility

In the Random Waypoint mobility model, the nodes move in line segments. For each line segment, a random destination position (distributed uniformly over the scene) and a random speed are chosen. You can define a speed as a variate from which a new value will be drawn for each line segment; it is customary to specify it as `uniform(minSpeed, maxSpeed)`. When the node reaches the target position, it waits for the time `waitTime`, which can also be defined as a variate. After this time, the algorithm calculates a new random position, etc.

GaussMarkovMobility

The Gauss-Markov model contains a tuning parameter that controls the randomness in the movement of the node. Let the magnitude and direction of the speed of the node at the nth time step be s_n and d_n. The next speed and

direction are computed as

$$s_{n+1} = \alpha s_n + (1-\alpha)\bar{s} + \sqrt{(1-\alpha^2)}s_{x_n}$$

$$d_{n+1} = \alpha d_n + (1-\alpha)\bar{d} + \sqrt{(1-\alpha^2)}d_{x_n}$$

where $\bar{s}$ and $\bar{d}$ are constants representing the mean value of speed and direction as $n \to \infty$; and s_{x_n} and d_{x_n} are random variables with a Gaussian distribution.

Totally random walk (Brownian motion) is obtained by setting $\alpha = 0$, while $\alpha = 1$ results in linear motion.

To ensure that the node does not remain at the boundary of the constraint area for a long time, the mean value of the direction ($\bar{d}$) is modified as the node enters the margin area. For example, at the right edge of the area it is set to 180 degrees, so the new direction is away from the edge.

MassMobility

This is a random mobility model for a mobile host with mass. It is the one used in [**?**].

> "An MH moves within the room according to the following pattern. It moves along a straight line for a certain period of time before it makes a turn. This moving period is a random number, normally distributed with the average of 5 seconds and standard deviation of 0.1 seconds. When it makes a turn, the new direction (angle) in which it will move is a normally distributed random number with an average equal to the previous direction and standard deviation of 30 degrees. Its speed is also a normally distributed random number, with a controlled average, ranging from 0.1 to 0.45 (unit/sec), and standard deviation of 0.01 (unit/sec). A new such random number is picked as its speed when it makes a turn. This pattern of mobility is intended to model node movement during which the nodes have momentum, and thus do not start, stop, or turn abruptly. When it hits a wall, it reflects off the wall at the same angle; in our simulated world, there is little other choice."

This implementation can be parameterized further, via the `changeInterval` and `angleDelta` parameters (and the `speed` parameter, which can be given as a variate). The parameters described above correspond to the following settings:

- changeInterval = normal(5, 0.1)
- angleDelta = normal(0, 30)
- speed = normal(avgSpeed, 0.01)

3D (spatial) movement is also supported, via the `rotationAxisAngle` parameter.

ChiangMobility

Implements Chiang's random walk movement model ([**?**]). In this model, the state of the mobile node in each direction (x and y) can be:

- 0: the node stays in its current position
- 1: the node moves forward
- 2: the node moves backward

The (i, j) element of the state transition matrix determines the probability that the state changes from i to j:

$$\begin{pmatrix} 0 & 0.5 & 0.5 \\ 0.3 & 0.7 & 0 \\ 0.3 & 0 & 0.7 \end{pmatrix}$$

SatelliteMobility

`SatelliteMobility` models the motion of an Earth-orbiting satellite. The orbit is given as a TLE (two-line element) set, the standard format published for tracked objects, and is propagated with an embedded SGP4 analytical model. At each update the model computes the satellite state in the TEME inertial frame, rotates it to the Earth-fixed frame using the Greenwich sidereal time, and converts it to WGS84 geodetic coordinates (latitude, longitude, altitude), which are finally mapped to the scene by the `Wgs84EcefGeographicCoordinateSystem`.

The TLE data is supplied as raw text in the `tleData` parameter, typically loaded with `readFile("satellites.tle")`. When the data contains more than one object, the satellite is selected by `satelliteIndex`, `satelliteName`, or `satelliteCatalogNumber`. The `epoch` parameter is the absolute UTC time (ISO-8601, e.g. `2026-06-02T00:00:00Z`) mapped to simulation time zero; if left empty, the satellite's own TLE epoch is used. The `attitudeMode` parameter controls the body orientation by aiming the body +X axis: `earth_fixed` (held fixed in the rotating ECEF frame), `star_fixed` (held fixed in the inertial frame, i.e. star-pointing), `nadir` (toward the Earth center), `zenith` (radially outward), or `velocity` (along the velocity vector).

Each satellite is an ordinary network node with a `SatelliteMobility` submodule. In a multi-satellite scenario the nodes typically share their settings (TLE data, update interval, attitude) via a wildcard, while the TLE selection (and, if needed, the epoch) is set individually per node.

Note

The Earth-orientation model is GMST-only: nutation, polar motion, and the UT1-UTC difference are neglected. This yields a ground-track error on the order of tens of meters, which is adequate for network simulation but not for precision orbit determination.

28.2.3 Replaying Trace Files

BonnMotionMobility

Uses the native file format of BonnMotion.

The file is a plain text file, where every line describes the motion of one host. A line consists of one or more (t, x, y) triplets of real numbers, like:

```
t1 x1 y1 t2 x2 y2 t3 x3 y3 t4 x4 y4 ...
```

The meaning is that the given node gets to (xk, yk) at tk. There's no separate notation for wait, so x and y coordinates will be repeated there.

Ns2MotionMobility

Nodes are moving according to the trace files used in NS2. The trace file has this format:

```
# '#' starts a comment, ends at the end of line
$node_(<id>) set X_ <x> # sets x coordinate of the node identified by <id>
$node_(<id>) set Y_ <y> # sets y coordinate of the node identified by <id>
$node_(<id>) set Z_ <z> # sets z coordinate (ignored)
$ns at $time "$node_(<id>) setdest <x> <y> <speed>" # at $time start moving
towards <x>,<y> with <speed>
```

The `Ns2MotionMobility` module has the following parameters:

- `traceFile` the Ns2 trace file
- `nodeId` node identifier in the trace file; -1 gets substituted by the parent module's index
- `scrollX`, `scrollY` user specified translation of the coordinates

ANSimMobility

It reads XML trace files from the ANSim Tool. The nodes are moving along linear segments described by an XML trace file conforming to this DTD:

```
<!ELEMENT mobility (position_change*)>
<!ELEMENT position_change (node_id, start_time, end_time, destination)>
<!ELEMENT node_id (#PCDATA)>
<!ELEMENT start_time (#PCDATA)>
```

(continues on next page)

(continued from previous page)

```
<!ELEMENT end_time (#PCDATA)>
<!ELEMENT destination (xpos, ypos)>
<!ELEMENT xpos (#PCDATA)>
<!ELEMENT ypos (#PCDATA)>
```

Parameters of the module:

- `ansimTrace` the trace file
- `nodeId` the `node_id` of this node, -1 gets substituted by the parent module's index

Note

The `AnsimMobility` module processes only the `position_change` elements, and it ignores the `start_time` attribute. It starts the move on the next segment immediately.

28.2.4 TurtleMobility

The `TurtleMobility` module can be parameterized by a script file containing LOGO-style movement commands in XML format. The content of the XML file should conform to the DTD in the `TurtleMobility.dtd` file in the source tree.

The file contains `movement` elements, each describing a trajectory. The `id` attribute of the `movement` element can be used to refer to the movement from the ini file using the syntax:

```
**.mobility.turtleScript = xmldoc("turtle.xml", "movements//movement[@id='1']")
```

The motion of the node is composed of uniform linear segments. The `movement` elements may contain the following commands as elements (names in parens are recognized attribute names):

- `repeat(n)` repeats its content n times, or indefinitely if the `n` attribute is omitted.
- `set(x,y,speed,angle,borderPolicy)` modifies the state of the node. `borderPolicy` can be `reflect`, `wrap`, `placerandomly`, or `error`.
- `forward(d,t)` moves the node for `t` time or to the `d` distance with the current speed. If both `d` and `t` are given, then the current speed is ignored.
- `turn(angle)` increases the angle of the node by `angle` degrees.
- `moveto(x,y,t)` moves to point `(x,y)` in the given time. If t is not specified, it is computed from the current speed.
- `moveby(x,y,t)` moves by offset `(x,y)` in the given time. If t is not specified, it is computed from the current speed.
- `wait(t)` waits for the specified amount of time.

Attribute values must be given without physical units, distances are assumed to be given as meters, time intervals in seconds, and speeds in meters per seconds. Attributes can contain expressions that are evaluated each time the command is executed. The limits of the constraint area can be referenced as `$MINX`, `$MAXX`, `$MINY`, and `$MAXY`. Random number distributions generate a new random number when evaluated, so the script can describe random as well as deterministic scenarios.

To illustrate the usage of the module, we show how some mobility models can be implemented as scripts.

RectangleMobility:

```
<movement>
    <set x="$MINX" y="$MINY" angle="0" speed="10"/>
    <repeat>
        <repeat n="2">
```

(continues on next page)

(continued from previous page)

```
            <forward d="$MAXX-$MINX"/>
            <turn angle="90"/>
            <forward d="$MAXY-$MINY"/>
            <turn angle="90"/>
        </repeat>
    </repeat>
</movement>
```

Random Waypoint:

```
<movement>
    <repeat>
        <set speed="uniform(20,60)"/>
        <moveto x="uniform($MINX,$MAXX)" y="uniform($MINY,$MAXY)"/>
        <wait t="uniform(5,10)"/>
    </repeat>
</movement>
```

MassMobility:

```
<movement>
    <repeat>
        <set speed="uniform(10,20)"/>
        <turn angle="uniform(-30,30)"/>
        <forward t="uniform(0.1,1)"/>
    </repeat>
</movement>
```

CHAPTER

TWENTYNINE

MODELING POWER CONSUMPTION

29.1 Overview

Modeling power consumption becomes more and more important with the increasing number of embedded devices and the upcoming Internet of Things. Mobile personal medical devices, large-scale wireless environment monitoring devices, electric vehicles, solar panels, low-power wireless sensors, etc., require paying special attention to power consumption. High-fidelity simulation of power consumption allows designing power-sensitive routing protocols, MAC protocols with power management features, etc., which in turn results in more energy-efficient devices.

In order to help the modeling process, the power model is separated from other simulation models in the INET Framework. This separation makes the power model extensible, and it also allows easy experimentation with alternative implementations. In a nutshell, the power model consists of the following components:

- energy consumption models
- energy generation models
- temporary energy storage models

The power model elements fall into two categories, which have `Ep` and `Cc` as part of their names:

- `Ep` models are simpler and deal with energy and power quantities.
- `Cc` models are more realistic and deal with charge, current, and voltage quantities.

The following sections provide a brief overview of the power model.

29.2 Energy Consumer Models

Energy consumer models describe the energy consumption of devices over time. For example, energy is consumed when a transceiver transmits or receives a signal, when a CPU forwards a packet, or when a display is turned on.

In INET, an energy consumer model is an OMNeT++ simple module that implements the energy consumption of software processes or hardware devices over time. The main purpose of these models is to provide the power or current consumption for the current simulation time. Most often, energy consumers are included as submodules in the compound module of the hardware devices or software components.

INET provides only a few built-in energy consumer models:

- `AlternatingEpEnergyGenerator` is a trivial example of an energy/power-based statistical energy consumer model.
- `StateBasedEpEnergyConsumer` is a transceiver energy consumer model based on the radio mode and transmission/reception states.

To simulate power consumption in a wireless network, the energy consumer model type must be configured for the transceivers. The following example demonstrates how to configure the power consumption parameters for a transceiver energy consumer model:

```
*.host[*].wlan[*].radio.energyConsumer.typename = "StateBasedEpEnergyConsumer"
*.host[*].wlan[*].radio.energyConsumer.sleepPowerConsumption = 0.1mW
*.host[*].wlan[*].radio.energyConsumer.receiverIdlePowerConsumption = 2mW
*.host[*].wlan[*].radio.energyConsumer.receiverBusyPowerConsumption = 5mW
*.host[*].wlan[*].radio.energyConsumer.receiverReceivingPowerConsumption = 10mW
*.host[*].wlan[*].radio.energyConsumer.transmitterIdlePowerConsumption = 2mW
*.host[*].wlan[*].radio.energyConsumer. \
   transmitterTransmittingPowerConsumption = 100mW # continue previous line
```

29.3 Energy Generator Models

Energy generator models describe the energy generation of devices over time. For example, a solar panel produces energy based on time, the panel's location on the globe, its orientation towards the sun, and the actual weather conditions. Energy generators connect to an energy storage that absorbs the generated energy.

In INET, an energy generator model is an OMNeT++ simple module that implements the energy generation of a hardware device using a physical phenomenon over time. The main purpose of these models is to provide the power or current generation for the current simulation time. Most often, energy generation models are included as submodules in network nodes.

INET provides only one trivial energy/power-based statistical energy generator model called `AlternatingEpEnergyGenerator`. The following example shows how to configure its power generation parameters:

```
*.host[*].energyGenerator.typename = "AlternatingEpEnergyGenerator"
*.host[*].energyGenerator.energySinkModule = "^.energyStorage" # module ref.
*.host[*].energyGenerator.powerGeneration = 1mW
*.host[*].energyGenerator.sleepInterval = exponential(10s) # random intervals
*.host[*].energyGenerator.generationInterval = exponential(10s)
```

29.4 Energy Storage Models

Electronic devices that are not connected to an external power source must contain some component to store energy. For example, a mobile phone contains an electrochemical battery that provides energy for its display, CPU, and communication devices. It might also absorb energy produced by a solar panel installed on its display or by a portable charger plugged into the wall socket.

In the INET Framework, an energy storage model is an OMNeT++ simple module that models the physical phenomena used to store energy produced by generators and provide energy for consumers. The main purpose of these models is to compute the amount of available energy or charge at the current simulation time. They maintain a set of connected energy consumers and energy generators, their respective total power consumption, and total power generation.

INET contains a few built-in energy storage models:

- `IdealEpEnergyStorage` is an idealistic model with infinite energy capacity and infinite power flow.
- `SimpleEpEnergyStorage` is a non-trivial model that integrates the difference between the total consumed power and the total generated power over time.
- `SimpleCcBattery` is a more realistic charge/current-based battery model that uses a charge independent ideal voltage source and internal resistance.

The following example shows how to configure a simple energy storage model:

```
*.host[*].energyStorage.typename = "SimpleEpEnergyStorage"
*.host[*].energyStorage.nominalCapacity = 0.05J # maximum capacity
*.host[*].energyStorage.initialCapacity = uniform(0J, nominalCapacity)
```

29.5 Energy Management Models

SimpleEpEnergyManagement

```
*.host[*].energyManagement.typename = "SimpleEpEnergyManagement"
*.host[*].energyManagement.nodeStartCapacity = 0.025J # start threshold
*.host[*].energyManagement.nodeShutdownCapacity = 0J # shutdown threshold
```

CHAPTER

THIRTY

NETWORK EMULATION

30.1 Motivation

In INET, the term *emulation* is used in a broad sense to describe a system which is partially implemented in the real world and partially in simulation. Emulation is often utilized for testing and validating a simulation model with its real-world counterparts, or in a reverse scenario, during the development of a real-world protocol implementation or application, for testing in a simulated environment. It may also be necessitated by the fact that some parts of the system only exist in the real world or only as simulation models.

INET provides modules that act as bridges between the simulated and real domains; hence, it is possible to leave one part of the simulation unchanged, while simply extracting the other into the real world. Several setups are possible where one can take advantage of the emulation capabilities of INET:

- simulated node in a real network
- a simulated subnet in a real network
- real-world node in a simulated network
- simulated protocol in a real network node
- real application in a simulated network node
- etc.

Some example scenarios:

- A simulated component, such as an app or a routing protocol, can be run on nodes of an actual ad-hoc network. This setup would allow testing the behavior of the component under real-life conditions.
- The interoperability of a simulated protocol with its real-world counterparts can be tested.
- As a means of implementing hybrid simulation, the real network (or a single host OS) may contain several network emulator devices or simulations running in emulation mode. Such a setup provides a relatively easy way of connecting heterogeneous simulators/emulators with each other, eliminating the need for HLA or a custom interoperability solution.

30.2 Overview

For the simulation to act as a network emulator, two major problems need to be solved. On one hand, the simulation must run in real time, or the real clock must be configured according to the simulation time (synchronization). On the other hand, the simulation must be able to communicate with the real world (communication). This is achieved in INET as follows:

- Synchronization:
  - `RealTimeScheduler:` It is a socket-aware real-time scheduler class responsible for synchronization. Using this method, the simulation is run according to real time.
- Communication:

– The interface between the real (an interface of the OS) and the simulated parts of the model is represented by *Ext* modules, with names beginning with the `Ext~` prefix in the simulation (`ExtLowerUdp`, `ExtUpperEthernetInterface`, etc.).

  Ext modules communicate both internally in the simulation and externally in the host OS. Packets sent to these modules in the simulation will be sent out on the host OS interface, and packets received by the host OS interface (or rather, the appropriate subset of them) will appear in the simulation as if received on an `Ext~` module.

  There are several possible ways to maintain the external communication:

  * File
  * Pipe
  * Socket
  * Shared memory
  * TUN/TAP interfaces
  * Message Passing Interface (MPI)

Note

It is probably needless to say, but the simulation must be fast enough to be able to keep up with real time. That is, its relative speed compared to real time (the simsec/sec value) must be >>1. (Under Qtenv, this can usually only be achieved in Express mode.)

30.3 Preparation

There are a few things that need to be arranged before simulations can be successfully run in network emulation mode.

First, network emulation is a separate *project feature* that needs to be enabled before it can be used. (Project features can be reviewed and changed in the *Project* | *Project Features...* dialog in the IDE.)

Also, in order to be able to send packets through raw sockets, applications require special permissions. There are two ways to achieve this under Linux.

The suggested solution is to use setcap to set the application permissions:

```
$ sudo setcap cap_net_raw,cap_net_admin=eip /path/to/opp_run
$ sudo setcap cap_net_raw,cap_net_admin=eip /path/to/opp_run_dbg
$ sudo setcap cap_net_raw,cap_net_admin=eip /path/to/opp_run_release
```

This solution makes running the examples from the IDE possible. Alternatively, the application can be started with root privileges from the command line:

```
$ sudo `inet_dbg -p -u Cmdenv`
```

Note

In any case, it's generally a bad idea to start the IDE as superuser. Doing so may silently change the file ownership for certain IDE configuration files, and it may prevent the IDE from starting up for the normal user afterwards.

30.4 Configuring

Here we demonstrate one configuration example where the network nodes contain a `ExtLowerEthernetInterface`.

INET nodes such as `StandardHost` and `Router` can be configured to have `ExtLowerEthernetInterface` modules. The simulation may contain several nodes with external interfaces, and one node may also have several external interfaces.

Note

This is one of the many possible setups. Using other components than `ExtLowerEthernetInterface`, nodes may be split into simulated and real parts at any layer, and either the upper or the lower part may be real. See the Showcases for demonstration of some of these use cases.

A network node can be configured to have an external interface in the following way:

```
**.host1.numEthInterfaces = 1
**.host1.eth[0].typename = "ExtLowerEthernetInterface"
```

Also, the simulation must be configured to run under the control of the appropriate real-time scheduler class:

```
scheduler-class = "inet::RealTimeScheduler"
```

`ExtLowerEthernetInterface` has two important parameters that need to be configured. The `device` parameter should be set to the name of the real (or virtual) interface on the host OS. The `namespace` parameter can be set to utilize the network namespace functionality of Linux operating systems.

An example configuration:

```
**.numEthInterfaces = 1
**.eth[0].device = "veth0" # or "eth0" for example
**.eth[0].namespace = "host0" # optional
**.eth[0].mtu = 1500B
```

Let us examine the paths outgoing and incoming packets take and the necessary configuration requirements to make them work. We assume IPv4 as network layer protocol, but the picture does not change much with other protocols. We assume the external interface is named `eth[0]`.

30.4.1 Outgoing path

The network layer of the simulated node routes datagrams to its `eth[0]` external interface.

For that to happen, the routing table needs to contain an entry where the interface is set to `eth[0]`. Such entries are not created automatically; one needs to add them to the routing table explicitly, e.g. by using an `Ipv4NetworkConfigurator` and an appropriate XML file.

Another point is that if the packet comes from a local app (and from another simulated node), it needs to have a source IP address assigned. There are two ways for that to happen. If the sending app specified a source IP address, that will be used. Otherwise, the IP address of the `eth[0]` interface will be used, but for that, the interface needs to have an IP address at all. The MAC and IP address of external interfaces are automatically copied between the real and simulated counterparts.

Once in `eth[0]`, the datagram is serialized. Serialization is a built-in feature of INET packets. (Packets, or rather, packet chunks have multiple alternative representations, i.e. C++ object and serialized form, and conversion between them is transparent.)

The result of serialization is a byte array, which is written into a raw socket with a `sendto` system call.

The packet will then travel normally in the real network to the destination address.

30.4.2 Incoming path

First of all, packets intended to be received by the simulation need to find their way to the correct interface of the host that runs the simulation. For that, IP addresses of simulated hosts must be routable in the real network and routed to the selected interface of the host OS. (On Linux, for example, this can be achieved by adding static routes with the command.)

As packets are received by the interface of the host OS, they are handed over to the simulation. The packets are received from the raw socket with a `recv` system call. After deserialization, they pop out of `eth[0]` and they are sent up to the network layer. The packets are routed to the simulated destination host in the normal way.

CHAPTER

THIRTYONE

SCENARIO SCRIPTING

31.1 Overview

The INET Framework contains scripting support to help scenarios that cannot be adequately described using static configuration. Actions can be scheduled to be carried out at specified simulation times, such as changing a parameter value, changing the bit error rate of a connection, removing or adding connections, removing or adding routes in a routing table, shutting down or crashing routers, etc. The aim is usually to observe transient behavior caused by the changes.

INET supports the following built-in actions:

- Create or delete module
- Create or delete connection
- Set module or channel parameter
- Initiate lifecycle operation (startup, shutdown, crash) on a network node or part of it

31.2 ScenarioManager

The `ScenarioManager` module type is used for setting up and controlling simulation experiments. In typical usage, it has only one instance in the network:

```
network Test {
    submodules:
        scenarioManager: ScenarioManager;
        ...
}
```

`ScenarioManager` executes a script specified in XML. It has a few built-in commands, while other commands are dispatched (in C++) to be carried out by other simple modules.

An example script:

```
<scenario>
    <set-param t="10" module="host[1].mobility" par="speed" value="5"/>
    <set-param t="20" module="host[1].mobility" par="speed" value="30"/>
    <at t="50">
        <set-param module="host[2].mobility" par="speed" value="10"/>
        <set-param module="host[3].mobility" par="speed" value="10"/>
        <connect src-module="host[2]" src-gate="ppp[0]"
                 dest-module="host[1]" dest-gate="ppp[0]"
                 channel-type="ned.DatarateChannel">
            <param name="datarate" value="10Mbps" />
            <param name="delay" value="0.1us" />
        </connect>
    </at>
```

(continues on next page)

(continued from previous page)

```
    <at t="60">
        <disconnect src-module="host[2]" src-gate="ppp[0]" />
    </at>
</scenario>
```

The above script probably does not need much explanation.

The built-in commands of `ScenarioManager` are: `<connect>`, `<disconnect>`, `<create-module>`, `<delete-module>`, `<initiate>`, `<shutdown>`, `<startup>`, `<crash>`, `<set-param>`, `<set-channel-attr>`, `<at>`.

All commands have a `t` attribute which carries the simulation time at which the command has to be carried out. Several commands can be grouped to be carried out at the same simulation time using `<at>`, and then only the `<at>` command needs to have a `t` attribute.

More information can be found in the `ScenarioManager` documentation.

The script is usually placed in a separate file and specified like this:

```
*.scenarioManager.script = xmldoc("scenario.xml")
```

Short scripts can also be written inline:

```
*.scenarioManager.script = xml("<x><shutdown t='2s' module='Router2'/></x>")
```

CHAPTER

THIRTYTWO

MODELING NODE FAILURES

32.1 Overview

Simulation is often used to study the effects of unexpected events like router crashes on the network. In order to accommodate such scenarios, INET supports *lifecycle modeling* of network nodes. The up/down status of a node is changed via lifecycle operations.

INET supports the following lifecycle operations:

- *Startup* represents the process of booting up or starting a network node after a shutdown or crash operation.
- *Shutdown* represents the process of orderly shutting down a network node.
- *Crash* represents the process of crashing a network node. The difference between *crash* and *shutdown* is that for a crash, the network node will not perform a graceful shutdown (e.g. routing protocols will not have a chance to notify peers about broken routes).

In a real-life router or other network node, a crash or shutdown and subsequent restart affects all parts of the system. All non-persistent information is lost. Protocol states are reset, various tables are cleared, connections are broken or torn down, applications restart, and so on.

Mimicking this behavior in simulation does not come for free; it needs to be explicitly programmed into each affected component. Here are some examples of how INET components react to a *crash* lifecycle event:

- `Tcp` forgets all open connections and sockets.
- `Ipv4` clears the fragmentation reassembly buffers and pending packets.
- `Ipv4RoutingTable` clears the route table.
- `EthernetCsmaMacPhy` and other MAC protocols clear their queues and reset their state associated with the current transmission(s).
- `Ospfv2` clears its full state.
- `UdpBasicApp`, `TcpSessionApp`, and other applications reset their state and stop/restart their timers.
- `EthernetSwitch`, `AccessPoint`, and other L2 bridging devices clear their MAC address tables.

While down, network interfaces, and components in general, ignore (discard) messages sent to them.

Lifecycle operations are currently instantaneous; they complete in zero simulation time. The underlying framework would allow for modeling them as processes that take place in some finite (nonzero) simulation time, but this possibility is currently not in use.

It is also possible to simulate a crash or shutdown of part of a node (certain protocols or interfaces only). Such scenarios would correspond to, for example, the crash of an OSPF daemon on a real OS.

Some energy-related INET components trigger node shutdown or crash under certain conditions. For example, a node will crash when it runs out of power (e.g. its battery depletes); see the chapter on power consumption modeling *Modeling Power Consumption* for details.

In the following sections, the INET components that participate in lifecycle modeling are outlined and a usage example is shown.

32.2 NodeStatus

Node models contain a `NodeStatus` module that keeps track of the status of the node (up, down, etc.) for other modules, and also displays it in the GUI as a small overlay icon.

The `NodeStatus` module is declared conditionally (so that it is only created in simulations that need it) like this:

```
status: NodeStatus if hasStatus;
```

If lifecycle modeling is required, the following line must be added to the INI file to ensure that nodes have status modules:

```
**.hasStatus = true
```

32.3 Scripting

Lifecycle operations can be triggered from C++ code or from scripts. INET supports scripting via the `ScenarioManager` NED type, described in chapter *Scenario Scripting*. Here is an example script that shuts down a router at simulation time 2s and starts it up again at time 8s:

```
<scenario>
  <initiate t="2s" module="Router2" operation="shutdown"/>
  <initiate t="8s" module="Router2" operation="startup"/>
</scenario>
```

The `module` attribute should point to the module (host, router, network interface, protocol, etc.) to be operated on. The `operation` attribute should contain the operation to perform: `"shutdown"`, `"crash"`, or `"startup"`. `t` is the simulation time at which the operation should be initiated.

An alternative, shorter form is to use <`shutdown`>, <`crash`>, or <`startup`> elements instead of the `operation` attribute:

```
<scenario>
  <shutdown t="2s" module="Router2"/>
  <startup  t="8s" module="Router2"/>
</scenario>
```

CHAPTER

THIRTYTHREE

PACKET FILTER EXPRESSIONS

A packet filter expression is used to evaluate a packet and return a boolean value. It is used in several modules, such as packet filters, packet classifiers, packet schedulers, various visualizers, and so on. The filter expression can be either a simple string literal or a full-blown NED expression.

The string variant is basically a pattern that is matched against the name of the packet. It can still contain boolean operators such as and/or/not, but it cannot refer to any part of the packet other than its name.

For example, the following pattern matches all packets that have a name starting with "ping" and do not end with "reply":

```
"ping* and not *reply"
```

The expression variant is evaluated by the module for each packet as needed. The expression can contain all NED expression syntax, along with implicitly defined variables shown below. These implicitly defined variables can refer to the packet itself, a protocol-specific chunk, a chunk with a specific chunk type, or a packet tag with a specific type.

The following list provides a few examples:

`expr(hasBitError)` matches packets with a bit error

`expr(name == 'P1')` matches packets that have 'P1' as their name

`expr(name =~ 'P*')` matches packets that have a name starting with 'P'

`expr(totalLength >= 100B)` matches packets that are longer than 100 bytes

Implicitly defined variables can be utilized to check the presence of a chunk of a specific protocol or of a chunk of a specific type:

`expr(udp != null)` matches packets that have at least one UDP protocol chunk

`expr(has(udp))` shorthand for the above

`expr(has(udp[0]))` same as above using indexing

`expr(has(UdpHeader))` matches packets that have at least one chunk with the UdpHeader type

`expr(has(UdpHeader[0]))` same as above using indexing

`expr(has(ipv4))` matches packets that have at least one IPv4 protocol chunk

`expr(has(ipv4[0]))` a packet may contain multiple protocol headers; they can be indexed

`expr(has(Ipv4Header))` matches packets that have at least one chunk with the Ipv4Header type

`expr(has(Ipv4Header[0]))` same as above using indexing

`expr(has(ethernetmac))` matches packets that have at least one Ethernet MAC protocol chunk

`expr(has(ethernetmac[0]))` same as above using indexing

`expr(has(ethernetmac[1]))` this would most likely match the Ethernet FCS chunk

`expr(has(EthernetMacHeader))` matches packets that have at least one chunk with the EthernetMacHeader type

`expr(has(EthernetMacHeader[0]))` same as above using indexing

The expression can also refer to fields of chunks:

`expr(ipv4.destAddress.getInt() == 0x0A000001)` matches packets with a specific binary IPv4 destination address

`expr(ipv4.destAddress.str() == '10.0.0.1')` same as above using strings

`expr(ipv4.destAddress.str() =~ '10.0.0.*')` matches packets that have an IPv4 destination address starting with 10.0.0.

`expr(udp.destPort == 42)` matches packets if the UDP destination port number equals 42

Expressions can also be combined using boolean operators:

```
expr(name == 'P1' && totalLength == 128B && ipv4.destAddress.str() == '10.0.0.1' &&
udp.destPort == 42)
```

CHAPTER

THIRTYFOUR

COLLECTING RESULTS

The following sections introduce the INET specific concepts and features for collecting simulation results. For more information in general on collecting statistics, please refer to the OMNeT++ manual.

34.1 Recording Statistics

Most INET modules are already equipped with the collection of various statistics, defined using the `@statistic` properties in the corresponding NED files. By default, many of them are already configured to be automatically recorded as either scalars, vectors, or histograms depending on the typical usefulness of the statistic. Note that the default recording mode can be changed from INI files as described in the OMNeT++ manual.

If the default statistics provided by the INET modules are not sufficient, new modules can be derived using only NED files and new statistics can be added to them based on the signals emitted by the module. The emitted signals can be found as `@signal` properties declared in the corresponding NED files.

If even declaring new statistics isn't sufficient, new C++ classes can be derived from the module implementations and new signals and/or new statistic collection code can be added to them. This is the most cumbersome way to collect new results, but it's also the most expressive, allowing to collect any kind of statistic.

34.2 Measuring along Packet Flows

By default, INET statistics are only capable of collecting results based on the data that individual protocol modules and applications can access. Statistics are collected independently by each module in the network, making them less useful in complex scenarios. For example, a TCP protocol module that communicates with multiple other TCP modules cannot distinguish between the packets based on the path they took. To overcome this issue, INET introduces the notion of packet flows.

A packet flow is a logical classification of packets, identified by its name, over the whole network and over the duration of the whole simulation. Basically, at any given moment, any packet that is present anywhere in the network can be part of any number of packet flows. Packets may enter a packet flow and leave it multiple times. Different packet flows can overlap both in time and also along the network topology. Note that a packet flow doesn't necessarily have a single entry and a single exit point.

A packet flow is usually defined by active modules that classify certain packets (e.g., matching a filter) to be entering the flow, and similarly, other modules that decide when packets leave the flow. Both kinds of modules are inserted into the network for this specific purpose, usually at the network interface level. While a packet is being part of any number of packet flows, certain INET modules (e.g., queues) are going to automatically record certain events that happen with the packets (e.g., queueing).

So far, the notion of packet flows was introduced on the packet level. That is, at any given time, a packet is either completely part of a packet flow or not. In fact, this is a simplification for easier understanding. This approach is clearly not sufficient in the general case because packets can be fragmented and aggregated throughout the network, and they can traverse many different paths. Therefore, the packet flow membership is actually specified on a per-bit basis. The implementation takes care of efficiently representing this data, so the coherent parts of a packet that belong to the same packet flow are marked together.

Using the packet flow mechanism, one can easily measure the timing of various things that happen to a packet (or to a part of it). The following quantities are automatically measured (if requested) along packet flows:

- total elapsed time measured from entering the packet flow
- total time spent in queues (e.g., transmission queue)
- total delay spent in various non-queue modules (e.g., interframe gap)
- total time spent in various packet processing modules (e.g., packet server)
- total transmission time of transmitters
- total propagation time spent on the transmission medium

The collected timing data is attached to coherent regions of the packets while the packets are in the packet flow. The actual measurement, that is, collecting the statistical results, is usually done when the packets leave the packet flow.

If the timing statistics are not sufficient, it's also possible to collect all packet events that happen to packets in the packet flow. The following packet events can be automatically collected along packet flows:

- packet is enqueued in a packet queue (e.g., transmission queue)
- packet is delayed (e.g., interframe gap)
- packet is processed (e.g., packet server)
- packet is transmitted by a wired or wireless transmitter
- packet is propagated on a wired or wireless transmission medium

The collected packet event data structure is also attached to coherent regions of the packet while the packets are in the packet flow. To actually make a measurement, a new measurement module must be implemented which processes the collected data.

In order to configure one of the above packet flow measurements, the following modules can be used:

- `FlowMeasurementStarter`: classifies packets to enter packet flows and starts timing measurements or packet event collection.
- `FlowMeasurementRecorder`: completes timing measurements, collects statistics, and makes packets leave the packet flow.
- `MeasurementLayer`: can be added to network nodes and network interfaces for optional packet flow measurements.

34.3 Recording PCAP Traces

The easiest way to understand the behavior of a communication network on the network level is to look at the actual packets that are exchanged. INET supports recording such packet traces in the widely used PCAP and its more recent sibling, the PCAPng file formats. These file formats allow analyzing the network traffic using the well-known Wireshark packet analyzer.

All network nodes and network interfaces support the recording of incoming and outgoing traffic into PCAP files via optional `PcapRecorder` modules. By default, this module records all packets emitted with configured signals from its parent module. For example, a recorder module put in the network interface module records all traffic specific to that network interface, and similarly a recorder put in the network node records all traffic from the given node. It's also possible to put a PCAP recorder module on the network level to produce a PCAP file that contains all network traffic. If the traffic of more than one network interface is recorded into a single file, then the newer PCAPng file format must be used to also record the data of the corresponding network interfaces.

Recording PCAP traces also support using packet filters, which in turn allows one to produce multiple files for the same network interface containing different kinds of traffic.

34.4 Recording Routing Tables

Understanding the behavior of routing protocols, especially with respect to the dynamic state of all routing tables, in a complex communication network is a difficult thing to do. In order to ease this task, INET provides a special `RoutingTableRecorder` module which is capable of recording all network interface data along with all routes in all routing tables into a single log file.

The log file uses a simple text-based format. It contains one line for each network interface added, changed, or deleted, and it also contains one line for each route added, changed, or deleted in all network nodes. The resulting log file can be used, for example, to verify that certain changes don't affect the way routes are discovered in the network.

CHAPTER

THIRTYFIVE

VISUALIZATION

35.1 Overview

The INET Framework is capable of visualizing a wide range of events and conditions in the network: packet drops, data link connectivity, wireless signal path loss, transport connections, routing table routes, and many more. Visualization is implemented as a collection of configurable INET modules that can be added to simulations at will.

35.2 Visualizing Network Communication

35.2.1 Visualizing Packet Drops

Several network problems manifest themselves as excessive packet drops, such as poor connectivity, congestion, or misconfiguration. Visualizing packet drops helps in identifying such problems in simulations and reduces time spent on debugging and analysis. Poor connectivity in a wireless network can cause senders to drop unacknowledged packets after the retry limit is exceeded. Congestion can cause queues to overflow in a bottleneck router, resulting in packet drops.

Packet drops can be visualized by including a `PacketDropVisualizer` module in the simulation. The `PacketDropVisualizer` module indicates packet drops by displaying an animation effect at the node where the packet drop occurs. In the animation, a packet icon gets thrown out from the node icon and fades away.

The visualization of packet drops can be enabled with the visualizer's `displayPacketDrops` parameter. By default, packet drops at all nodes are visualized. This selection can be narrowed with the `nodeFilter`, `interfaceFilter`, and `packetFilter` parameters.

One can click on the packet drop icon to display information about the packet drop in the inspector panel.

Packets are dropped for the following reasons:

- Queue overflow
- Retry limit exceeded
- Unroutable packet
- Network address resolution failed
- Interface down

35.2.2 Visualizing Transport Path Activity

With INET simulations, it is often useful to be able to visualize network traffic. INET provides several visualizers for this task, operating at various levels of the network stack. One of such visualizers is `TransportRouteVisualizer` that provides graphical feedback about transport layer traffic.

`TransportRouteVisualizer` visualizes traffic that passes through the transport layers of two endpoints. Adding an `IntegratedVisualizer` is also an option, because it also contains a `TransportRouteVisualizer`. Transport path activity visualization is disabled by default, it can be enabled by setting the visualizer's `displayRoutes` parameter to true.

`TransportRouteVisualizer` observes packets that pass through the transport layer, i.e. carry data from/to higher layers.

The activity between two nodes is represented visually by a polyline arrow which points from the source node to the destination node. `TransportRouteVisualizer` follows packets throughout their path so that the polyline goes through all nodes which are the part of the path of packets. The arrow appears after the first packet has been received, then gradually fades out unless it is reinforced by further packets. Color, fading time and other graphical properties can be changed with parameters of the visualizer.

By default, all packets and nodes are considered for the visualization. This selection can be narrowed with the visualizer's `packetFilter` and `nodeFilter` parameters.

35.2.3 Visualizing Network Path Activity

Network layer traffic can be visualized by including a `NetworkRouteVisualizer` module in the simulation. Adding an `IntegratedVisualizer` module is also an option, because it also contains a `NetworkRouteVisualizer` module. Network path activity visualization is disabled by default, it can be enabled by setting the visualizer's `displayRoutes` parameter to true.

`NetworkRouteVisualizer` currently observes packets that pass through the network layer (i.e. carry data from/to higher layers), but not those that are internal to the operation of the network layer protocol. That is, packets such as ARP, although potentially useful, will not trigger the visualization.

The activity between two nodes is represented visually by a polyline arrow which points from the source node to the destination node. `NetworkRouteVisualizer` follows packet throughout its path so the polyline goes through all nodes that are part of the packet's path. The arrow appears after the first packet has been received, then gradually fades out unless it is reinforced by further packets. Color, fading time and other graphical properties can be changed with parameters of the visualizer.

By default, all packets and nodes are considered for the visualization. This selection can be narrowed with the visualizer's `packetFilter` and `nodeFilter` parameters.

35.2.4 Visualizing Data Link Activity

Data link activity (layer 2 traffic) can be visualized by adding a `DataLinkVisualizer` module to the simulation. Adding an `IntegratedVisualizer` module is also an option, because it includes a `DataLinkVisualizer` module. Data link visualization is disabled by default, it can be enabled by setting the visualizer's `displayLinks` parameter to true.

`DataLinkVisualizer` currently observes packets that pass through the data link layer (i.e. carry data from/to higher layers), but not those that are internal to the operation of the data link layer protocol. That is, frames such as ACK, RTS/CTS, Beacon or Authentication/Association frames of IEEE 802.11, although potentially useful, will not trigger the visualization. Visualizing such frames may be implemented in future INET revisions.

The activity between two nodes is represented visually by an arrow that points from the sender node to the receiver node. The arrow appears after the first packet has been received, then gradually fades out unless it is refreshed by further packets. The style, color, fading time and other graphical properties can be changed with parameters of the visualizer.

By default, all packets, interfaces and nodes are considered for the visualization. This selection can be narrowed to certain packets and/or nodes with the visualizer's `packetFilter`, `interfaceFilter`, and `nodeFilter` parameters.

35.2.5 Visualizing Physical Link Activity

Physical link activity can be visualized by including a `PhysicalLinkVisualizer` module in the simulation. Adding an `IntegratedVisualizer` module is also an option, as it also contains a `PhysicalLinkVisualizer` module. Physical link activity visualization is disabled by default, but it can be enabled by setting the visualizer's `displayLinks` parameter to true.

`PhysicalLinkVisualizer` observes frames that pass through the physical layer, meaning they are received correctly.

The activity between two nodes is visually represented by a dotted arrow that points from the sender node to the receiver node. The arrow appears after the first frame has been received and gradually fades out unless it is refreshed by further frames. Color, fading time, and other graphical properties can be changed with parameters of the visualizer.

By default, all packets, interfaces, and nodes are considered for the visualization. This selection can be narrowed with the visualizer's `packetFilter`, `interfaceFilter`, and `nodeFilter` parameters.

35.2.6 Visualizing Routing Tables

In a complex network topology, it is difficult to see how a packet would be routed because the relevant data is scattered among network nodes and hidden in their routing tables. INET contains support for visualization of routing tables and can display routing information graphically in a concise way. Using visualization, it is often possible to understand routing in a simulation without looking into individual routing tables. The visualization currently supports IPv4.

The `RoutingTableVisualizer` module (included in the network as part of `IntegratedVisualizer`) is responsible for visualizing routing table entries.

The visualizer basically annotates network links with labeled arrows that connect source nodes to next hop nodes. The module visualizes those routing table entries that participate in the routing of a given set of destination addresses, by default the addresses of all interfaces of all nodes in the network. That is, it selects the best (longest prefix) matching routes for all destination addresses from each routing table and shows them as arrows that point to the next hop. Note that one arrow might need to represent several routing entries, for example, when distinct prefixes are routed towards the same next hop.

Routing table entries are visually represented by solid arrows. An arrow going from a source node represents a routing table entry in the source node's routing table, and the endpoint node of the arrow is the next hop in the visualized routing table entry. By default, the routing entry is displayed on the arrows in the following format:

destination/mask -> gateway (interface)

The format can be changed by setting the visualizer's `labelFormat` parameter.

Filtering is also possible. The `nodeFilter` parameter controls which nodes' routing tables should be visualized (by default, all nodes), and the `destinationFilter` parameter selects the set of destination nodes to consider (again, by default all nodes).

The visualizer reacts to changes, such as when a routing protocol changes a routing entry or an IP address gets assigned to an interface by DHCP. The visualizer automatically updates the visualizations according to the specified filters. This is very useful for the simulation of mobile ad-hoc networks.

35.2.7 Displaying IP Addresses and Other Interface Information

In the simulation of complex networks, it is often useful to be able to display node IP addresses, interface names, etc. above the node icons or on the links. For example, when automatic address assignment is used in a hierarchical network (e.g. using `Ipv4NetworkConfigurator`), visual inspection can help to verify that the result matches the expectations. While it is true that addresses and other interface data can also be accessed in the GUI by diving into the interface tables of each node, that is tedious, and unsuitable for getting an overview.

The `InterfaceTableVisualizer` module (included in the network as part of `IntegratedVisualizer`) displays data about network nodes' interfaces. (Interfaces are contained in interface tables, hence the name.) By default, the visualization is turned off. When it is enabled using the `displayInterfaceTables` parameter, the default is that interface names, IP addresses and netmask length are displayed, above the nodes (for wireless interfaces) and on the links (for wired interfaces). By clicking on an interface label, details are displayed in the inspector panel.

The visualizer has several configuration parameters. The `format` parameter specifies what information is displayed about interfaces. It takes a format string, which can contain the following directives:

- %N: interface name
- %4: IPv4 address
- %6: IPv6 address

- %n: network address. This is either the IPv4 or the IPv6 address
- %l: netmask length
- %M: MAC address
- %: conditional newline for wired interfaces. The '' needs to be escaped with another '', i.e. '%\'
- %i and %s: the info() and str() functions for the networkInterface class, respectively

The default format string is `"%N %\\%n/%l"`, i.e. interface name, IP address and netmask length.

The set of visualized interfaces can be selected with the configurator's `nodeFilter` and `interfaceFilter` parameters. By default, all interfaces of all nodes are visualized, except for loopback addresses (the default for the `interfaceFilter` parameter is `"not lo\"`.)

It is possible to display the labels for wired interfaces above the node icons, instead of on the links. This can be done by setting the `displayWiredInterfacesAtConnections` parameter to false.

There are also several parameters for styling, such as color and font selection.

35.2.8 Visualizing IEEE 802.11 Network Membership

When simulating Wi-Fi networks that overlap in space, it is difficult to see which node is a member of which network. The membership may even change over time. It would be useful to be able to display the SSID above node icons.

IEEE 802.11 network membership can be visualized by including an `Ieee80211Visualizer` module in the simulation. Adding an `IntegratedVisualizer` is also an option, as it also contains an `Ieee80211Visualizer`. Displaying network membership is disabled by default, but it can be enabled by setting the visualizer's `displayAssociations` parameter to true.

The `Ieee80211Visualizer` displays an icon and the SSID above network nodes that are part of a Wi-Fi network. The icons are color-coded according to the SSID. The icon, colors, and other visual properties can be configured via parameters of the visualizer.

The visualizer's `nodeFilter` parameter selects which nodes' memberships are visualized. The `interfaceFilter` parameter selects which interfaces are considered in the visualization. By default, all interfaces of all nodes are considered.

35.2.9 Visualizing Transport Connections

In a large network with a complex topology, it can be difficult to determine which nodes are communicating and if any communication is occurring at all. Visualizing transport connections provides an easy way to obtain information about active transport connections in the network. It allows for easy identification and differentiation of connections based on their endpoints, providing a quick overview of the number of connections in individual nodes and the entire network.

The `TransportConnectionVisualizer` module (also part of `IntegratedVisualizer`) displays color-coded icons above the two endpoints of an active, established transport layer connection. The icons appear when the connection is established and disappear when it is closed. Multiple connections at a node result in multiple icons. Icons for the same connection have the same color at both endpoints. Additionally, letter codes can be displayed to help identify connections. Note that this visualizer does not display the paths of packets. For path visualization, refer to the `TransportRouteVisualizer` covered in the section *Visualizing Transport Path Activity*.

The visualization is disabled by default, but it can be enabled by setting the visualizer's `displayTransportConnections` parameter to true.

Connections to be visualized can be filtered using the `sourcePortFilter`, `destinationPortFilter`, `sourceNodeFilter`, and `destinationNodeFilter` parameters.

The icons and other visual properties can be customized using the visualizer's parameters.

35.3 Visualizing The Infrastructure

35.3.1 Visualizing the Physical Environment

The physical environment significantly impacts the communication of wireless devices. Objects like walls within buildings limit mobility and obstruct radio signals, often causing packet loss. To better understand simulations, it is crucial to visualize the physical objects present in the environment.

The visualization of physical objects is achieved using the `PhysicalEnvironmentVisualizer` (also part of `IntegratedVisualizer`). The objects themselves are provided by the PhysicalEnvironment module, defining their geometry, physical properties, and visual appearance in the XML configuration of the PhysicalEnvironment module.

The two-dimensional projection of physical objects is determined by the `SceneCanvasVisualizer` module. The default view is a top view (z-axis), but side views (x and y axes) or isometric and orthographic projections can also be configured.

For 3D rendering using the OpenSceneGraph (OSG) library, the simulation and OMNeT++ both need to be compiled with OSG support. The 3D view can be activated using the Qtenv toolbar.

35.3.2 Visualizing Node Mobility

In INET simulations, the movement of mobile nodes is often as important as the communication among them. However, as mobile nodes roam, it is often difficult to visually follow their movement. INET provides a visualizer that not only makes visually tracking mobile nodes easier but also indicates other properties like speed and direction.

Node mobility can be visualized by the `MobilityVisualizer` module (included in the network as part of `IntegratedVisualizer`). By default, mobility visualization is enabled, but it can be disabled by setting the `displayMobility` parameter to false.

By default, all mobilities are considered for the visualization. This selection can be narrowed with the visualizer's `moduleFilter` parameter.

The visualizer has several important features:

- Movement Trail: It displays a line along the recent path of movements. The trail gradually fades out as time passes. Color, trail length, and other graphical properties can be changed with parameters of the visualizer.
- Velocity Vector: Velocity is visually represented by an arrow. Its starting point is the node, and its direction coincides with the movement's direction. The arrow's length is proportional to the node's speed.
- Orientation Cone: Node orientation is represented as a 3D cone whose axis points along the node's facing direction, drawn as its (projected) base disc plus two side lines. The cone's angular width relative to a full circle is set by the `orientationPieSize` parameter (default 0.2) and its on-screen radius by `orientationPieRadius`. As a depth cue, the base disc is filled with `orientationFillColorUp` when the cone points toward the viewer and with `orientationFillColorDown` when it points away.

These features are disabled by default but can be enabled by setting the visualizer's `displayMovementTrails`, `displayVelocities`, and `displayOrientations` parameters to true.

On an equirectangular world map (see *Visualizing the World Map*), the velocity arrow and the orientation cone are direction indicators, and their appearance under the map's distortion is controlled by the `velocityProjection` and `orientationProjection` parameters: `projected` follows the full map distortion, `trueLength` keeps the true length along the (sheared) map heading, and `isometric` preserves both true heading and length. These parameters have no effect without a geographic projection.

35.4 Visualizing Satellite and Geographic Scenarios

INET can run global, map-based scenarios in which network nodes (satellites, ground stations) are positioned by geographic coordinates in a geocentric scene (see `Wgs84EcefGeographicCoordinateSystem`). Because such a scene spans the whole Earth, the default 2D scene view is not useful; INET provides several dedicated canvas visualizers for these scenarios: a world map background, a satellite visibility footprint overlay, and a sky view.

They all share a common map projection so that the nodes drawn by the ordinary visualizers (node icons, mobility trails, etc.) line up with the map.

35.4.1 Visualizing the World Map

The `GeoMapCanvasVisualizer` draws a 2D world map on the canvas: a background map image overlaid with an optional latitude/longitude graticule, and an optional absolute UTC wall-clock in the lower-left corner. The map image is configured with the `mapImage` parameter (an image name looked up on the image path), and the visible window is given by the `minLatitude`, `maxLatitude`, `minLongitude`, and `maxLongitude` parameters, sized to `mapWidth` by `mapHeight` pixels. The map uses an equirectangular (plate-carrée) projection, so the `mapImage` must be an equirectangular rendering of that window. The wall-clock is enabled by setting the `epoch` parameter to the absolute UTC time (ISO-8601) that corresponds to simulation time zero.

Crucially, this visualizer also configures the shared canvas projection from the same window, so that every other canvas visualizer that draws scene coordinates — the node icons of the `NetworkNodeVisualizer`, the markers, movement trails, velocity and orientation indicators of the `MobilityVisualizer`, and the footprint overlay below — appears at the correct place on the map. The map visualizer itself does not draw any per-node markers.

When the window covers only part of the globe (a regional map), the `clipFigures` parameter (enabled by default) clips those other visualizers' figures — node icons, mobility markers and trails, links, and routes — to the map area, so nothing is drawn projected outside the visible window.

35.4.2 Visualizing Satellite Visibility Footprints

The `GeoHorizonCanvasVisualizer` draws the visibility footprint of each tracked node on the map: the area on the Earth's surface from which the node is seen at or above a configurable elevation angle. This is the region within which a satellite is in view of a ground station. The footprint is recomputed every refresh from the node's current position.

The nodes to track are selected with the `moduleFilter` parameter, and their geographic positions are computed through the geocentric/ECEF coordinate system given by `coordinateSystemModule`. The visualizer takes its map window from the map visualizer named by its `mapVisualizer` parameter, so the footprint automatically lines up with the background and needs no separate window configuration. The minimum elevation angle is set with `elevationMask`.

35.4.3 Visualizing the Sky View

The `GeoSkyViewCanvasVisualizer` draws an azimuth/elevation polar plot, the "sky view", next to each observer node, showing where the target nodes currently are in the observer's sky. It is primarily intended for ground stations observing satellites.

The nodes that get a plot are selected with the `observerFilter` parameter, and the nodes drawn in each plot with the `targetFilter` parameter. For every observer-target pair, the visualizer computes the topocentric look angles (azimuth, elevation, and range) from their geographic positions.

In the plot, the center is the zenith (90 degrees elevation) and the outer rim is the horizon (0 degrees); North is up and East is to the right. Concentric rings mark the elevation, the N/E/S/W cardinal directions are drawn, and the band below the `elevationMask` parameter is shaded with a translucent mask. Each visible target is shown as a colored dot, optionally with its name and a fading "sky trail" of its recent path; a target that drops below the horizon is hidden.

CHAPTER

THIRTYSIX

INSTRUMENT FIGURES

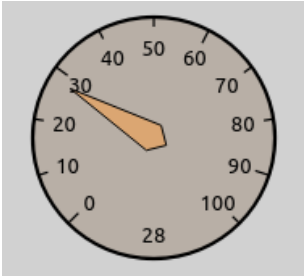
36.1 Overview

In complex simulations, there are usually several statistics that are vital for understanding what is happening inside the network. Although statistics can also be found and read in Qtenv's object inspector panel, it is often more convenient to display them directly on the top-level canvas in a graphical form. INET supports such visualization in the form of inspector figures that display various gauges and meters. This chapter covers the usage of instrument figures.

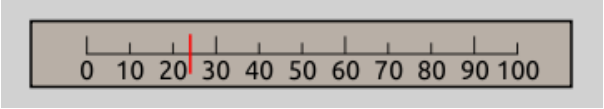
36.2 Instrument Types

Some of the instrument figure types available in INET are as follows:

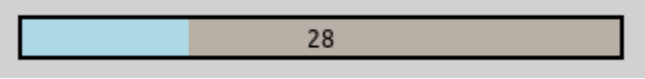
- *gauge:* A circular gauge similar to a speedometer or pressure indicator.



- *linearGauge:* A horizontal linear gauge similar to a VU meter.

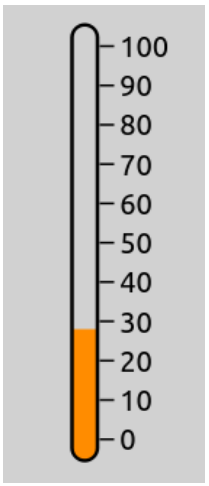


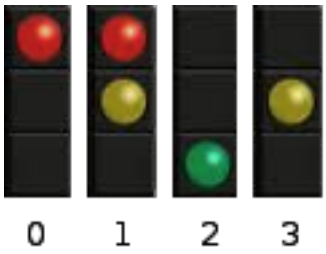
- *progressMeter:* A horizontal progress bar.



- *counter:* An integer counter.
- *thermometer:* A vertical meter visually similar to a real-life thermometer.
- *indexedImage:* A figure that displays one of several images: the first image for the value 0, the second image for the value of 1, and so on.

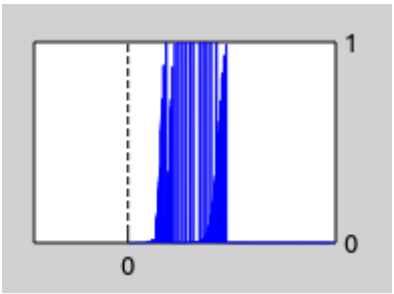








- *plot:* An XY chart that plots a statistic as a function of time.



36.3 Using Instrument Figures

Instrument figures visualize statistics derived from signals emitted by modules in the network. This statistic is declared in the NED file, with the `@statistic` property. The property's `source` attribute is an expression that specifies which signals to use from which modules, and the mathematical operations on them, to derive the statistic. The `record` attribute specifies where the values of the statistic are recorded. In the case of instrument figures, this is set to `figure`, i.e., `record=figure`. The `targetFigure` attribute selects which figure should display the statistic.

The instrument figure itself is specified in the NED file with the `@figure` property. The property's `type` attribute selects the type of instrument figure (`gauge`, `thermometer`, etc.), and the property's index (i.e., the figure name) should match the name given in the statistic's `targetFigure` attribute.

Here is an example NED file:

```
@statistic[numRcvdPk](source=count(client.app[0].rcvdPk); record=figure;␣
↪targetFigure=counter);
```

(continues on next page)

(continued from previous page)

```
@figure[numRcvdPkCounter](type=counter; pos=413,327; label="Packets received";␣
↪decimalPlaces=4);
```

This creates a figure named numRcvdPkCounter, which displays a counter. The statistic numRcvdPk counts the packets received by the client host's first application, and records it in the numRcvdPkCounter figure.

36.4 Instrument Figure Attributes

Instrument figures have various attributes that customize their position, size, appearance, label text and font, minimum and maximum values, and so on. The following list shows the attributes recognized by the various figure types (a similar list can be produced by running INET with the -h figures command-line option):

counter:
type, visible, zIndex, tooltip, tags, transform, backgroundColor, decimalPlaces, digitBackgroundColor, digitBorderColor, digitFont, digitColor, label, labelFont, labelColor, initialValue, pos, anchor, labelOffset

gauge:
type, visible, zIndex, tooltip, tags, transform, backgroundColor, needleColor, label, labelFont, labelColor, minValue, maxValue, tickSize, colorStrip, initialValue, pos, size, anchor, bounds, labelOffset

indexedImage:
type, visible, zIndex, tooltip, tags, transform, images, tintAmount, tintColor, opacity, interpolation, label, labelFont, labelColor, labelOffset, initialValue, pos, size, anchor, bounds

indicatorLabel:
type, visible, zIndex, tooltip, tags, transform, pos, anchor, text, font, color, opacity, halo, textFormat, initialValue

indicatorText:
type, visible, zIndex, tooltip, tags, transform, pos, anchor, text, font, color, opacity, halo, textFormat, initialValue

linearGauge:
type, visible, zIndex, tooltip, tags, transform, backgroundColor, needleColor, label, labelFont, labelColor, minValue, maxValue, tickSize, cornerRadius, initialValue, pos, size, anchor, bounds, labelOffset

plot:
type, visible, zIndex, tooltip, tags, transform, valueTickSize, timeWindow, timeTickSize, lineColor, minValue, maxValue, backgroundColor, label, labelOffset, labelColor, labelFont, numberSizeFactor, pos, size, anchor, bounds

progressMeter:
type, visible, zIndex, tooltip, tags, transform, backgroundColor, stripColor, cornerRadius, borderWidth, minValue, maxValue, text, textFont, textColor, label, labelOffset, labelFont, labelColor, initialValue, pos, size, anchor, bounds

thermometer:
type, visible, zIndex, tooltip, tags, transform, mercuryColor, label, labelFont, labelColor, minValue, maxValue, tickSize, initialValue, pos, size, anchor, bounds, labelOffset

CHAPTER

THIRTYSEVEN

APPENDIX: AUTHOR'S GUIDE

37.1 Overview

This chapter is intended for authors and contributors of this *INET User's Guide*, and covers the guidelines for deciding what type of content is appropriate for this *Guide* and what is not.

The main guiding principle is to avoid redundancy and duplication of information with other pieces of documentation, namely:

- Standards documents (RFCs, IEEE specifications, etc.) that describe protocols that INET modules implement;
- *INET Developer's Guide*, which is intended as a guide for anyone wishing to understand or modify the operation of INET's components at the C++ level;
- *INET Framework Reference*, directly generated from NED and MSG comments by OMNeT++ documentation generator;
- Showcases, tutorials, and simulation examples (`showcases/`, `tutorials/`, and `examples/` folders in the INET project)

The reasons to avoid duplication are as follows:

- It wastes readers' time if they have to skip information they have seen elsewhere.
- The text can easily become outdated as the INET Framework evolves.
- It requires extra effort for maintainers to keep all copies up to date.

37.2 Guidelines

37.2.1 Do Not Repeat the Standard

When describing a module that implements protocol X, avoid going into lengthy explanations of how protocol X works or what it does since this is usually better explained in the specification or books on protocol X. It is permissible to summarize the protocol's goal and principles in a short paragraph instead.

In particular, avoid describing the *protocol message format.* Although it may look nice and take up a lot of space, the same information can probably be found in various places on the internet.

37.2.2 Do Not Repeat NED

Module parameters, gate names, emitted signals, and collected statistics are formally part of the NED definitions, and there is no need to duplicate that information in this *Guide*.

Detailed information about the module, such as *usage details* and the list of *implemented standards,* should be covered in the module's NED documentation, not in this *Guide*.

37.2.3 No C++

Any content that only makes sense at the C++ level should be included in the *Developer's Guide* and should not be included in this *Guide*.

37.2.4 Keep Examples Short

When providing usage examples, keep them concise and to the point. Providing ini or NED file fragments a few lines in length is preferable to providing complete working examples.

Complete examples should be written up as showcases. A working simulation without much commentary should be placed under `examples`. A practical, potentially multi-step guide to using a nontrivial feature should be written up as a tutorial.

37.2.5 No Reference to Simulation Examples

Avoid referring to specific example simulations, showcases, or tutorials in the text because they might be renamed, moved, merged, or deleted. When this occurs, no one will consider updating the reference in the *Users Guide*.

37.2.6 What then?

Focus on providing a "big picture" of the models: what they are generally capable of, how the parts fit together, etc. Provide enough information that after a quick read, users can "bootstrap" themselves into creating their own simulations with the model. If users have questions afterwards, they can refer to the NED documentation (INET Reference) or, if necessary, delve into the C++ code to find the answers.

CHAPTER

THIRTYEIGHT

HISTORY

38.1 IPSuite to INET Framework (2000-2006)

The INET framework's predecessor was written by Klaus Wehrle, Jochen Reber, Dirk Holzhausen, Volker Boehm, Verena Kahmann, Ulrich Kaage, and others at the University of Karlsruhe during 2000-2001, under the name IPSuite.

The MPLS, LDP, and RSVP-TE models were built as an add-on to IPSuite during 2003 by Xuan Thang Nguyen (Xuan.T.Nguyen@uts.edu.au) and other students at the University of Technology, Sydney under the supervision of Dr. Robin Brown. The package consisted of around 10,000 LOCs and was published at http://charlie.it.uts.edu.au/tkaphan/xtn/capstone (now unavailable).

After a period of IPSuite being unmaintained, the development was taken over by Andras Varga in July 2003. Through a series of snapshot releases in 2003-2004, modules were completely reorganized, documented, and many of them were rewritten from scratch. The MPLS models (including RSVP-TE, LDP, etc.) were also refactored and merged into the codebase.

During 2004, a new, modular and extensible TCP implementation, application models, Ethernet implementation, and an all-in-one IP model were added to replace the earlier, modularized one.

The package was renamed INET Framework in October 2004.

Support for wireless and mobile networks was added during the summer of 2005 by using code from the Mobility Framework.

The MPLS models (including LDP and RSVP-TE) were revised and mostly rewritten from scratch by Vojta Janota in the first half of 2005 for his diploma thesis. After further refinements by Vojta, the new code was merged into the INET CVS in fall 2005 and eventually released in the March 2006 INET snapshot.

The OSPFv2 model was created by Andras Babos during 2004 for his diploma thesis, which was submitted early 2005. This work was sponsored by Andras Varga, using revenues from commercial OMNEST licenses. After several refinements and fixes, the code was merged into the INET Framework in 2005 and became part of the March 2006 INET snapshot.

The Quagga routing daemon was ported into the INET Framework also by Vojta Janota. This work was also sponsored by Andras Varga. During the fall of 2005 and the months after, ripd and ospfd were ported, and the methodology of porting was refined. Further Quagga daemons still remain to be ported.

Based on experience from the IPv6Suite (from Ahmet Sekercioglu's group at CTIE, Monash University, Melbourne) and IPv6SuiteWithINET (Andras's effort to refactor IPv6Suite and merge it with INET early 2005), Wei Yang Ng (Monash Uni) implemented a new IPv6 model from scratch for the INET Framework in 2005 for his diploma thesis, under guidance from Andras, who was visiting Monash between February and June 2005. This IPv6 model was first included in the July 2005 INET snapshot and gradually refined afterwards.

The SCTP implementation was contributed by Michael Tuexen, Irene Ruengeler, and Thomas Dreibholz.

Support for Sam Jensen's Network Simulation Cradle, which makes real-world TCP stacks available in simulations, was added by Zoltan Bojthe in 2010.

TCP SACK and New Reno implementation were contributed by Thomas Reschka.

Several other people have contributed to the INET Framework by providing feedback, reporting bugs, suggesting features, and contributing patches; their help is acknowledged here as well.