INET Framework for OMNeT++

Manual

Generated on September 22, 2014

Contents

Chapter 1

Introduction

1.1 What is INET Framework

INET Framework contains IPv4, IPv6, TCP, SCTP, UDP protocol implementations, and several application models. The framework also includes an MPLS model with RSVP-TE and LDP signalling. Link-layer models are PPP, Ethernet and 802.11. Static routing can be set up using network autoconfigurators, or one can use routing protocol implementations.

The INET Framework supports wireless and mobile simulations as well. Support for mobility and wireless communication has been derived from the Mobility Framework.

1.2 About the documentation

This manual is acompanied by a Reference generated from NED and MSG files using OMNeT++'s documentation generator, and the documentation of the underlying C++ classes, generated from the source files using Doxygen.

The C++ doc is generated in a way that it contains the full C++ source code as HTML pages. It is syntax highlighted, and variable and class names are hyperlinked and cross-referenced, which makes it convenient for exploring the code.

1.3 Contents of this Manual

todo...

Chapter 2

Using the INET Framework

2.1 Installation

To install the INET Framework, download the most recent INET Framework source release from the download link on the http://inet.omnetpp.org web site, unpack it, and follow the instructions in the INSTALL file in the root directory of the archive.

If you plan to simulate ad-hoc networks or you need more wireless link layer protocols than provided in INET, download and install the INETMANET source archive instead. (INETMANET aready contains a copy of the INET Framework.)

If you plan to make use of other INET extensions (e.g. HttpTools, VoipTools or TraCI), follow the installation instructions provided with them. If there are no install instructions, check if the archive contains a `.project` file. If it does, then the project can be imported into the IDE (use File > Import > General > Existing Project into workspace); make sure that the project is recognized as an OMNeT++ project (the Project Properties dialog contains an OMNeT++ page) and it lists the INET or INETMANET project as dependency (check the Project References page in the Project Properties dialog).

If the extension project contains no `.project` file, create an empty OMNeT++ project using the New OMNeT++ Project wizard in File > New, add the INET or INETMANET project as dependency using the Project References page in the Project Properties dialog, and copy the source files into the project.

After installation, run the example simulations to make sure everything works correctly. The INSTALL file also describes how to do that.

2.2 INET as an OMNeT++-based simulation framework

TODO what is documented where in the OMNeT++ manual (chapter, section title)

The INET Framework builds upon OMNeT++, and uses the same concept: modules that communicate by message passing. Hosts, routers, switches and other network devices are represented by OMNeT++ compound modules. These compound modules are assembled from simple modules that represent protocols, applications, and other functional units. A network is again an OMNeT++ compound module that contains host, router and other modules. The external interfaces of modules are described in NED files. NED files describe the parameters and gates (i.e. ports or connectors) of modules, and also the submodules and connections (i.e.

netlist) of compound modules.

Modules are organized into hierarchical *packages* that directly map to a folder tree, very much like Java packages. Packages in INET are organized roughly according to OSI layers; the top packages include `inet.applications`, `inet.transport`, `inet.networklayer`, and `inet.linklayer`. Other packages are `inet.base`, `inet.util`, `inet.world`, `inet.mobility` and `inet.nodes`. These packages correspond to the `src/applications/`, `src/transport/`, etc. directories in the INET source tree. (The `src/` directory corresponds to the `inet` package, as defined by the `src/package.ned` file.) Subdirectories within the top packages usually correspond to concrete protocols or protocol families. The implementations of simple modules are C++ classes with the same name, with the source files placed in the same directory as the NED file.

The `inet.nodes` package contains various pre-assembled host, router, switch, access point, and other modules, for example `StandardHost`, `Router` and `EtherSwitch` and `WirelessAP`. These compound modules contain some customization options via parametric submodule types, but they are not meant to be universal: it is expected that you will create your own node models for your particular simulation scenarios.

Network interfaces (Ethernet, IEEE 802.11, etc) are usually compound modules themselves, and are being composed of a queue, a MAC, and possibly other simple modules. See `EthernetInterface` as an example.

Not all modules implement protocols. There are modules which hold data (for example `RoutingTable`), facilitate communication of modules (`NotificationBoard`), perform autoconfiguration of a network (`FlatNetworkConfigurator`), move a mobile node around (for example `ConstSpeedMobility`), and perform housekeeping associated with radio channels in wireless simulations (`ChannelControl`).

Protocol headers and packet formats are described in message definition files (msg files), which are translated into C++ classes by OMNeT++'s *opp_msgc* tool. The generated message classes subclass from OMNeT++'s `cPacket` or `cMessage` classes.

The internal structure of compound modules such as host and router models can be customized in several ways. The first way is the use of *gate vectors* and *submodule vectors*. The sizes of vectors may come from parameters or derived by the number of external connections to the module. For example, one can have an Ethernet switch model that has as many ports as needed, i.e. equal to the number of Ethernet devices connected to it.

The second way of customization is *parametric types*, that is, the type of a submodule (or a channel) may be specified as a string parameter. For example, the relay unit inside an Ethernet switch has several alternative implementations, each one being a distinct module type. The switch model contains a parameter which allows the user to select the appropriate relay unit implementation.

A third way of customizing modules is *inheritance*: a derived module may add new parameters, gates, submodules or connections, and may set inherited unassigned parameters to specific values.

2.3 Creating and Running Simulations

To create a simulation, you would write a NED file that contains the network, i.e. routers, hosts and other network devices connected together. You can use a text editor or the IDE's graphical editor to create the network.

Modules in the network contain a lot of unassigned parameters, which need to be assigned

before the simulation can be run.[1] The name of the network to be simulated, parameter values and other configuration option need to be specified in the `omnetpp.ini` file.[2]

`omnetpp.ini` contains parameter assignments as *key=value* lines, where each key is a wildcard pattern. The simulator matches these wildcard patterns against full path of the parameter in the module tree (something like `ThruputTest.host[2].tcp.nagleEnabled`), and value from the first match will be assigned for the parameter. If no matching line is found, the default value in the NED file will be used. (If there is no default value either, the value will be interactively prompted for, or, in case of a batch run, an error will be raised.)

There are two kinds of wildcards: a single asterisk `*` matches at most one component name in the path string, while double asterisk `**` may match multiple components. Technically: `*` never matches a dot or a square bracket (`.`, `[`, `]`), while `**` can match any of them. Patterns are also capable of expressing index ranges (`**.host[1..3,5,8].tcp.nagleEnabled`) and ranges of numbers embedded in names (`**.switch{2..3}.relayUnitType`).

OMNeT++ allows several configurations to be put into the `omnetpp.ini` file under `[Config <name>]` section headers, and the right configuration can be selected via command-line options when the simulation is run. Configurations can also build on each other: `extends=<name>` lines can be used to set up an inheritance tree among them. This feature allows minimizing clutter in ini files by letting you factor out common parts. (Another ways of factoring out common parts are ini file inclusion and specifying multiple ini files to a simulation.) Settings in the `[General]` section apply to all configurations, i.e. `[General]` is the root of the section inheritance tree.

Parameter studies can be defined by specifying multiple values for a parameter, with the `${10,50,100..500 step 100, 1000}` syntax; a repeat count can also be specified.

how to run;

C++ -> dll (opp_run) or exe

2.4 Result Collection and Analysis

how to anaylize results

how to configure result collection

2.5 Setting up wired network simulations

For an introduction, in this section we show you how to set up simulations of wired networks using PPP or Ethernet links with autoconfigured static IP routing. (If your simulation involves more, such as manually configured routing tables, dynamic routing, MPLS, or IPv6 or other features and protocols, you'll find info about them in later chapters.)

Such a network can be assembled using the predefined `StandardHost` and `Router` modules. For automatic IP address assignment and static IP routing we can use the `FlatNetworkConfigurator` utility module.

ethg, pppg; automatically expand (++)

todo which modules are needed into it, what they do, etc.

how to add apps, etc

[1] The simulator can interactively ask for parameter values, but this is not very convenient for repeated runs.
[2] This is the default file name; using other is also possible.

2.5.1 Modeling Link and Node Failures

todo

Some modules have only one instance, at global network level:

`FlatNetworkConfigurator` assigns IP addresses to hosts and routers, and sets up static routing.

`ScenarioManager` makes simulations scriptable. Modules can be made to support scripting by implementing the `IScriptable` C++ interface.

`ChannelControl` is required for wireless simulations. It keeps track of which nodes are within interference distance of other nodes.

2.5.2 Specifying IP (IPv6) addresses in module parameters

In INET, TCP, UDP and all application layer modules work with both IPv4 and IPv6. Internally they use the `IPvXAddress` C++ class, which can represent both IPv4 and IPv6 addresses.

Most modules use the `IPAddressResolver` C++ class to resolve addresses specified in module parameters in omnetpp.ini. `IPAddressResolver` accepts the following syntax:

- 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)"`

2.6 Setting up wireless network simulations

todo which modules are needed into it, what they do, etc.

2.7 Setting up ad-hoc network simulations

todo which modules are needed into it, what they do, etc.

2.8 Emulation

2.9 Packet traces

Recording packet traces

Traffic generation using packet traces

2.10 Developing New Protocols

where to put the source files: you can copy and modify the INET framework (fork it) in the hope that you'll contribute back the changes; or you can develop in a separate project (create new project in the IDE; mark INET as referenced project)

NED and source files in the same folder; examples under examples/; etc.

for details, read the OMNeT++ manual and the following chapters of this manual

todo...

Chapter 3

Node Architecture

3.1 Overview

This chapter describes the architecture of INET host and router models.

Hosts and routers in the INET Framework are OMNeT++ compound modules that are composed of the following ingredients:

- **Interface Table (`InterfaceTable`)**. This module contains the table of network interfaces (eth0, wlan0, etc) in the host. Interfaces are registered dynamically during the initialization phase by modules that represent network interface cards (NICs). Other modules access interface information via a C++ class interface.
- **Routing Table (`RoutingTable`)**. This module contains the IPv4 routing table. It is also accessed from other via a C++ interface. The interface contains member functions for adding, removing, enumerating and looking up routes, and finding the best matching route for a given destination IP address. The IP module calls uses the latter function for routing packets, and routing protocols such as OSPF or BGP call the route manipulation methods to add and manage discovered routes. For IPv6 simulations, `RoutingTable` is replaced (or co-exists) with a `RoutingTable6` module; and for Mobile IPv6 simulations (xMIPv6 project [TODO]), possibly with a `BindingCache` module as well.
- **Notification Board (`NotificationBoard`)**. This module makes it possible for several modules to communicate in a publish-subscribe manner. Notifications include change notifications ("routing table changed") and state changes ("radio became idle").
- **Mobility module**. In simulations involving node mobility, this module is responsible for moving around the node in the simulated "playground." A mobility module is also needed for wireless simulations even if the node is stationery, because the mobility module stores the node's location, needed to compute wireless transmissions. Different mobility models (Random Walk, etc.) are supported via different module types, and many host models define their mobility submodules with parametric type so that the mobility model can be changed in the configuration (`"mobility: <mobilityType> like IMobility"`).
- **NICs**. Network interfaces are usually compound modules themselves, composed of a queue and a MAC module (and in the case of wireless NICs, a radio module or modules). Examples are `PPPInterface`, `EthernetInterface`, and WLAN interfaces such as `Ieee80211NicSTA`. The queue submodule stores packets waiting for transmission, and

it is usually defined as having parametric type as it has multiple implementations to accomodate different needs (`DropTailQueue`, `REDQueue`, `DropTailQoSQueue`, etc.) Most MACs also have an internal queue to allow operation without an external queue module, the advantage being smaller overhead. The NIC's entry in the host's `InterfaceTable` is usually registered by the MAC module at the beginning of the simulation.

- **Network layer**. Modules that represent protocols of the network layer are usually grouped into a compound module: `NetworkLayer` for IP, and `NetworkLayer6` for IPv6. `NetworkLayer` contains the modules `IP`, `ARP`, `ICMP` and `ErrorHandling`. The `IP` module performs IP encapsulation/decapsulation and routing of datagrams; for the latter it accesses the C++ function call interface of the `RoutingTable`. Packet forwarding can be turned on/off via a module parameter. The `ARP` module is put into the path of packets leaving the network layer towards the NICs, and performs address resolution for interfaces that need it (e.g. Ethernet). `ICMP` deals with sending and receiving ICMP packets. The `ErrorHandling` module receives and logs ICMP error replies. The IPv6 network layer, `NetworkLayer6` contains the modules `IPv6`, `ICMPv6`, `IPv6NeighbourDiscovery` and `IPv6ErrorHandling`. For Mobile IPv6 simulations (xMIPv6 project [TODO]), `NetworkLayer6` is extended with further modules.

- **Transport layer protocols**. Transport protocols are represented by modules connected to the network layer; currently TCP, UDP and SCTP are supported. TCP has several implementations: `TCP` is the OMNeT++ native implementation; the `TCP_lwip` module wraps the lwIP TCP stack [TODO]; and the `TCP_NSC` module wraps the Network Simulation Cradle library [TODO]. For this reason, the `tcp` submodule is usually defined with a parametric submodule type ("`tcp: <tcpType> like ITCP`"). UDP and SCTP are implemented by the `UDP` and `SCTP` modules, respectively.

- **Applications**. Application modules typically connect to TCP and/or UDP, and model the user behavior as well as the application program (e.g. browser) and application layer protocol (e.g. HTTP). For convenience, `StandardHost` supports any number of UDP, TCP and SCTP applications, their types being parametric ("`tcpApp[numTcpApps]: <tcpAppType> like TCPApp; udpApp[numUdpApps]: <udpAppType> like UDPApp; ...`"). This way the user can configure applications entirely from `omnetpp.ini`, and does not need to write a new NED file every time different applications are needed in a host model. Application modules are typically not present in router models.

- **Routing protocols**. Router models typically contain modules that implement routing protocols such as OSPF or BGP. These modules are connected to the TCP and/or the UDP module, and manipulate routes in the `RoutingTable` module via C++ function calls.

- **MPLS modules**. Additional modules are needed for MPLS simulations. The `MPLS` module is placed between the network layer and NICs, and implements label switching. `MPLS` requires a `LIB` module (Label Information Base) to be present in the router which it accesses via C++ function calls. MPLS control protocol implementations (e.g. the `RSVP` module) manage the information in `LIB` via C++ calls.

- **Relay unit**. Ethernet (and possibly other) switch models may contain a relay unit, which switches frames among Ethernet (and other IEEE 802) NICs. Concrete relay unit types include `MACRelayUnitPP` and `MACRelayUnitNP`, which differ in their performance models.

- **Battery module**. INET extensions uses for wireless sensor networks (WSNs) may add a battery module to the node model. The battery module would keep track of energy consumption. A battery module is provided e.g. by the INETMANET project.

The StandardHost and Router predefined NED types are only one possible example of host/router models, and they do not contain all the above components; for specific simulations it is a perfectly valid approach to create custom node models.

Most modules are optional, i.e. can be left out of hosts or other node models when not needed in the given scenario. For example, switch models do not need a network layer, a routing table or interface table; it may need a notification board though. Some NICs (e.g. EtherMAC) can be used without and interface table and queue models as well.

The notification board (NotificationBoard) and the interface table (InterfaceTable) will be described later in this chapter. Other modules are covered in later chapters, i.e. RoutingTable in the IPv4 chapter.

3.2 Addresses

The INET Framework uses a number of C++ classes to represent various addresses in the network. These classes support initialization and assignment from binary and string representation of the address, and accessing the address in both forms. (Storage is in binary form). They also support the "unspecified" special value (and the isUnspecified() method) that corresponds to the all-zeros address.

- MACAddress represents a 48-bit IEEE 802 MAC address. The textual notation it understands and produces is hex string.
- IPAddress represents a 32-bit IPv4 address. It can parse and produce textual representations in the "dotted decimal" syntax.
- IPv6Address represents a 128-bit IPv6 address. It can parse and produce address strings in the canonical (RFC 3513) syntax.
- IPvXAddress is conceptually a union of a IPAddress and IPv6Address: an instance stores either an IPv4 address or an IPv6 address. IPvXAddress is mainly used in the transport layer and above to abstract away network addresses. It can be assigned from both IPAddress and IPv6Address, and can also parse string representations of both. The isIPv6(), get4() and get6() methods can be used to access the value.

TODO: Resolving addresses with IPAddressResolver

3.3 The Notification Board

The NotificationBoard module allows publish-subscribe communication among modules within a host. Using NotificationBoard, modules can notify each other about events such as routing table changes, interface status changes (up/down), interface configuration changes, wireless handovers, changes in the state of the wireless channel, mobile node position changes, etc. NotificationBoard acts as a intermediary between the module where the events occur and modules which are interested in learning about those events.

NotificationBoard has exactly one instance within a host or router model, and it is accessed via direct C++ method calls (not message exchange). Modules can *subscribe* to categories of changes (e.g. “routing table changed” or “radio channel became empty”). When such a change occurs, the corresponding module (e.g. the RoutingTable or the physical layer module) will

let NotificationBoard know, and it will disseminate this information to all interested modules.

Notification events are grouped into *categories*. Examples of categories are: NF_HOSTPOSITION_UPDATED, NF_RADIOSTATE_CHANGED, NF_PP_TX_BEGIN, NF_PP_TX_END, NF_IPv4_ROUTE_ADDED, NF_BEACON_LOST, NF_NODE_FAILURE, NF_NODE_RECOVERY, etc. Each category is identified by an integer (right now it's assigned in the source code via an enum, in the future we'll convert to dynamic category registration).

To trigger a notification, the client must obtain a pointer to the NotificationBoard of the given host or router (explained later), and call its fireChangeNotification() method. The notification will be delivered to all subscribed clients immediately, inside the fireChangeNotification() call.

Clients that wish to receive notifications should implement (subclass from) the INotifiable interface, obtain a pointer to the NotificationBoard, and subscribe to the categories they are interested in by calling the subscribe() method of the NotificationBoard. Notifications will be delivered to the receiveChangeNotification() method of the client (redefined from INotifiable).

In cases when the category itself (an int) does not carry enough information about the notification event, one can pass additional information in a data class. There is no restriction on what the data class may contain, except that it has to be subclassed from cObject, and of course producers and consumers of notifications should agree on its contents. If no extra info is needed, one can pass a NULL pointer in the fireChangeNotification() method.

A module which implements INotifiable looks like this:

```
class Foo : public cSimpleModule, public INotifiable {
    ...
    virtual void receiveChangeNotification(int category, const cObject *details) {..}
    ...
};
```

Note: cObject was called cPolymorphic in earlier versions of OMNeT++. You may occasionally still see the latter name in source code; it is an alias (typedef) to cObject.

Obtaining a pointer to the NotificationBoard module of that host/router:

```
NotificationBoard *nb; // this is best made a module class member
nb = NotificationBoardAccess().get();  // best done in initialize()
```

TODO how to fire a notification

3.4 The Interface Table

The InterfaceTable module holds one of the key data structures in the INET Framework: information about the network interfaces in the host. The interface table module does not send or receive messages; other modules access it using standard C++ member function calls.

3.4.1 Accessing the Interface Table

If a module wants to work with the interface table, first it needs to obtain a pointer to it. This can be done with the help of the InterfaceTableAccess utility class:

```
IInterfaceTable *ift = InterfaceTableAccess().get();
```

`InterfaceTableAccess` requires the interface table module to be a direct child of the host and be called "`interfaceTable`" in order to be able to find it. The `get()` method never returns `NULL`: if it cannot find the interface table module or cannot cast it to the appropriate C++ type (`IInterfaceTable`), it throws an exception and stop the simulation with an error message.

For completeness, `InterfaceTableAccess` also has a `getIfExists()` method which can be used if the code does not require the presence of the interface table. This method returns `NULL` if the interface table cannot be found.

Note that the returned C++ type is `IInterfaceTable`; the initial "`I`" stands for "interface". `IInterfaceTable` is an abstract class interface that `InterfaceTable` implements. Using the abstract class interface allows one to transparently replace the interface table with another implementation, without the need for any change or even recompilation of the INET Framework.

3.4.2 Interface Entries

Interfaces in the interface table are represented with the `InterfaceEntry` class. `IInterfaceTable` provides member functions for adding, removing, enumerating and looking up interfaces.

Interfaces have unique names and interface IDs; either can be used to look up an interface (IDs are naturally more efficient). Interface IDs are invariant to the addition and removal of other interfaces.

Data stored by an interface entry include:

- *name* and *interface ID* (as described above)
- *MTU*: Maximum Transmission Unit, e.g. 1500 on Ethernet
- several flags:
  - *down*: current state (up or down)
  - *broadcast*: whether the interface supports broadcast
  - *multicast* whether the interface supports multicast
  - *pointToPoint*: whether the interface is point-to-point link
  - *loopback*: whether the interface is a loopback interface
- *datarate* in bit/s
- *link-layer address* (for now, only IEEE 802 MAC addresses are supported)
- *network-layer gate index*: which gate of the network layer within the host the NIC is connected to
- *host gate IDs*: the IDs of the input and output gate of the host the NIC is connected to

Extensibility. You have probably noticed that the above list does not contain data such as the IP or IPv6 address of the interface. Such information is not part of `InterfaceEntry` because we do not want `InterfaceTable` to depend on either the IPv4 or the IPv6 protocol implementation; we want both to be optional, and we want `InterfaceTable` to be able to support possibly other network protocols as well.

Thus, extra data items are added to `InterfaceEntry` via extension. Two kinds of extensions are envisioned: extension by the link layer (i.e. the NIC), and extension by the network layer protocol:

- **NICs** can extend interface entries via C++ class inheritance, that is, by simply subclassing `InterfaceEntry` and adding extra data and functions. This is possible because NICs create and register entries in `InterfaceTable`, so in their code one can just write `new MyExtendedInterfaceEntry()` instead of `new InterfaceEntry()`.
- **Network layer protocols** cannot add data via subclassing, so composition has to be used. `InterfaceEntry` contains pointers to network-layer specific data structures. Currently there are four pointers: one for IPv4 specific data, one for IPv6 specific data, and two additional ones that are unassigned. The four data objects can be accessed with the following `InterfaceEntry` member functions: `ipv4Data()`, `ipv6Data()`, `protocol3Data()`, and `protocol4Data()`. They return pointers of the types `IPv4InterfaceData`, `IPv6InterfaceData` and `InterfaceProtocolData` (2x), respectively. For illustration, `IPv4InterfaceData` is installed onto the interface entries by the `RoutingTable` module, and it contains data such as the IP address of the interface, the netmask, link metric for routing, and IP multicast addresses associated with the interface. A protocol data pointer will be `NULL` if the corresponding network protocol is not used in the simulation; for example, in IPv4 simulations only `ipv4Data()` will return a non-`NULL` value.

3.4.3 Interface Registration

Interfaces are registered dynamically in the initialization phase by modules that represent network interface cards (NICs). The INET Framework makes use of the multi-stage initialization feature of OMNeT++, and interface registration takes place in the first stage (i.e. stage 0).

Example code that performs interface registration:

```
void PPP::initialize(int stage)
{
    if (stage==0) {
        ...
        interfaceEntry = registerInterface(datarate);
    ...
}

InterfaceEntry *PPP::registerInterface(double datarate)
{
    InterfaceEntry *e = new InterfaceEntry();

    // interface name: NIC module's name without special characters ([])
    e->setName(OPP_Global::stripnonalnum(getParentModule()->getFullName()).c_str());

    // data rate
    e->setDatarate(datarate);

    // generate a link-layer address to be used as interface token for IPv6
    InterfaceToken token(0, simulation.getUniqueNumber(), 64);
    e->setInterfaceToken(token);
```

```
    // set MTU from module parameter of similar name
    e->setMtu(par("mtu"));

    // capabilities
    e->setMulticast(true);
    e->setPointToPoint(true);

    // add
    IInterfaceTable *ift = InterfaceTableAccess().get();
    ift->addInterface(e, this);

    return e;
}
```

3.4.4 Interface Change Notifications

`InterfaceTable` has a change notification mechanism built in, with the granularity of interface entries.

Clients that wish to be notified when something changes in `InterfaceTable` can subscribe to the following notification categories in the host's `NotificationBoard`:

- **`NF_INTERFACE_CREATED`**: an interface entry has been created and added to the interface table
- **`NF_INTERFACE_DELETED`**: an interface entry is going to be removed from the interface table. This is a pre-delete notification so that clients have access to interface data that are possibly needed to react to the change
- **`NF_INTERFACE_CONFIG_CHANGED`**: a configuration setting in an interface entry has changed (e.g. MTU or IP address)
- **`NF_INTERFACE_STATE_CHANGED`**: a state variable in an interface entry has changed (e.g. the up/down flag)

In all those notifications, the data field is a pointer to the corresponding `InterfaceEntry` object. This is even true for `NF_INTERFACE_DELETED` (which is actually a pre-delete notification).

3.5 Initialization Stages

Node architecture makes it necessary to use multi-stage initialization. What happens in each stage is this:

In stage 0, interfaces register themselves in `InterfaceTable` modules

In stage 1, routing files are read.

In stage 2, network configurators (e.g. `FlatNetworkConfiguration`) assign addresses and set up routing tables.

In stage 3, TODO...

In stage 4, TODO...

...

The multi-stage initialization process itself is described in the OMNeT++ Manual.

3.6 Communication between protocol layers

In the INET Framework, when an upper-layer protocol wants to send a data packet over a lower-layer protocol, the upper-layer module just sends the message object representing the packet to the lower-layer module, which will in turn encapsulate it and send it. The reverse process takes place when a lower layer protocol receives a packet and sends it up after decapsulation.

It is often necessary to convey extra information with the packet. For example, when an application-layer module wants to send data over TCP, some connection identifier needs to be specified for TCP. When TCP sends a segment over IP, IP will need a destination address and possibly other parameters like TTL. When IP sends a datagram to an Ethernet interface for transmission, a destination MAC address must be specified. This extra information is attached to the message object to as *control info*.

Control info are small value objects, which are attached to packets (message objects) with its `setControlInfo()` member function. Control info only holds auxiliary information for the next protocol layer, and is not supposed to be sent over the network to other hosts and routers.

3.7 Publish-Subscribe Communication within Nodes

The `NotificationBoard` module makes it possible for several modules to communicate in a publish-subscribe manner. For example, the radio module (`Ieee80211Radio`) fires a *"radio state changed"* notification when the state of the radio channel changes (from TRANSMIT to IDLE, for example), and it will be delivered to other modules that have previously subscribed to that notification category. The notification mechanism uses C++ functions calls, no message sending is involved.

The notification board submodule within the host (router) must be called `notificationBoard` for other modules to find it.

3.8 Network interfaces

todo...

3.9 The wireless infrastructure

todo...

3.10 NED Conventions

3.10.1 The @node Property

By convention, compound modules that implement network devices (hosts, routers, switches, access points, base stations, etc.) are marked with the `@node` NED property. As node models may themselves be hierarchical, the `@node` property is used by protocol implementations and other simple modules to determine which ancestor compound module represents the physical network node they live in.

3.10.2 The @labels Module Property

The `@labels` property can be added to modules and gates, and it allows the OMNeT++ graphical editor to provide better editing experience. First we look at `@labels` as a module property.

`@labels(node)` has been added to all NED module types that may occur on network level. When editing a network, the graphical editor will NED types with `@labels(node)` to the top of the component palette, allowing the user to find them easier.

Other labels can also be specified in the `@labels(...)` property. This has the effect that if one module with a particular label has already been added to the compound module being edited, other module types with the same label are also brought to the top of the palette. For example, `EtherSwitch` is annotated with `@labels(node,ethernet-node)`. When you drop an `EtherSwitch` into a compound module, that will bring `EtherHost` (which is also tagged with the `ethernet-node` label) to the top of the palette, making it easier to find.

```
module EtherSwitch
{
    parameters:
        @node();
        @labels(node,ethernet-node);
        @display("i=device/switch");
    ...
}
```

Module types that are already present in the compound module also appear in the top part of the palette. The reason is that if you already added a `StandardHost`, for example, then you are likely to add more of the same kind. Gate labels (see next section) also affect palette order: modules which can be connected to modules already added to the compound module will also be listed at the top of the palette. The final ordering is the result of a scoring algorithm.

3.10.3 The @labels Gate Property

Gates can also be labelled with `@labels()`; the purpose is to make it easier to connect modules in the editor. If you connect two modules in the editor, the gate selection menu will list gate pairs that have a label in common.

TODO screenshot

For example, when connecting hosts and routers, the editor will offer connecting Ethernet gates with Ethernet gates, and PPP gates with PPP gates. This is the result of gate labelling like this:

```
module StandardHost
```

```
{
    ...
    gates:
        inout pppg[] @labels(PPPFrame-conn);
        inout ethg[] @labels(EtherFrame-conn);
    ...
}
```

Guidelines for choosing gate label names: For gates of modules that implement protocols, use the C++ class name of the packet or acompanying control info (see later) associated with the gate, whichever applies; append `/up` or `/down` to the name of the control info class. For gates of network nodes, use the class names of packets (frames) that travel on the corresponding link, with the `-conn` suffix. The suffix prevents protocol-level modules to be promoted in the graphical editor palette when a network is edited.

Examples:

```
simple TCP like ITCP
{
    ...
    gates:
        input appIn[] @labels(TCPCommand/down);
        output appOut[] @labels(TCPCommand/up);
        input ipIn @labels(TCPSegment,IPControlInfo/up);
        output ipOut @labels(TCPSegment,IPControlInfo/down);
        input ipv6In @labels(TCPSegment,IPv6ControlInfo/up);
        output ipv6Out @labels(TCPSegment,IPv6ControlInfo/down);
}
```

```
simple PPP
{
    ...
    gates:
        input netwIn;
        output netwOut;
        inout phys @labels(PPPFrame);
}
```

Chapter 4

Point-to-Point Links

4.1 Overview

The INET Framework contains an implementation of the Point-to-Point Protocol as described in RFC1661 with the following limitations:

- There are no LCP messages for link configuration, link termination and link maintenance. The link can be configured by setting module parameters.
- PFC and ACFC are not supported, the PPP frame always contains the 1-byte Address and Control fields and a 2-byte Protocol field.
- PPP authentication is not supported
- Link quality monitoring protocols are not supported.
- There are no NCP messages, the network layer protocols are configured by other means.

The modules of the PPP model can be found in the `inet.linklayer.ppp` package:

`PPP` This simple module 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 complementing the `PPP` module with an output queue. It implements the `IWiredNic` interface. Input and output hooks can be configured for further processing of the network messages.

4.2 PPP frames

According to RFC1662 the PPP frames contain the following fields:

Flag 01111110	Address 11111111	Control 00000011
Protocol 8/16 bits	Information *	Padding *
FCS 16/32 bits	Flag 01111110	Inter-frame Fill or next Address

The corresponding message type in the INET framework is `PPPFrame`. It contains the Information field as an encapsulated `cMessage` object. The Flag, Address and Control fields are omitted from `PPPFrame` because they are constants. The FCS field is omitted because no CRC computed during the simulation, the bit error attribute of the `cMessage` used instead. The Protocol field is omitted because the protocol is determined from the class of the encapsulated message.

The length of the PPP frame is equal to the length of the encapsulated datagram plus 7 bytes. This computation assumes that

- there is no inter-octet time fill, so only one Flag sequence needed per frame
- padding is not applied
- PFC and ACFC compression is not applied
- FCS is 16 bit
- no escaping was applied

4.3 PPP module

The PPP module receives packets from the upper layer in the `netwIn` gate, encapsulates them into `PPPFrame`s, and send it to the physical layer through the `phys` gate. The `PPPFrame`s received from the `phys` gate are decapsulated 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 `IOutputQueue`) 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 named txQueue to queue up packets wainting for transmission. Conceptually txQueue is of inifinite size, but for better diagnostics one can specify a hard limit in the `txQueueLimit` 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 dinamically. 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 transmits 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.

The node containing the PPP module must also contain a `NofiticationBoard` component. Notifications are sent when transmission of a new PPP frame started (`NF_PP_TX_BEGIN`), finished (`NF_PP_TX_END`) or when a PPP frame received (`NF_PP_RX_END`).

The PPP component is the source of the following signals:

- **txState** state of the link (0=idle,1=busy)
- **txPkBytes** number of bytes transmitted
- **rxPkBytesOk** number of bytes received successfully
- **droppedPkBytesBitError** number of bytes received in erronous frames
- **droppedPkBytesIfaceDown** number of bytes dropped because the link is down
- **rcvdPkBytesFromHL** number of bytes received from the the upper layer
- **passedUpPkBytes** number of bytes sent to the the upper layer

These signals are recorded as statistics (sum, count and vector), so they can be analyzed after the simulation.

When the simulation is executed with the graphical user interface the module displays useful statistics. If the link is operating, the datarate and number of received, sent and dropped messages show in the tooltip. When the link is broken, the number of dropped messages is displayed. The state of the module is indicated by the color of the module icon and the connection (yellow=transmitting).

4.4 PPPInterface module

The `PPPInterface` is a compound module implementing the `IWiredNic` 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 `queueType` parameter. It can be set to `NoQueue` or to a module type implementing the `IOutputQueue` 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 abstract that usage by adding `IHook` components to the network input and output of the `PPP` component. Any number of hook 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 5

The Ethernet Model

5.1 Overview

Variations: 10Mb/s ethernet, fast ethernet, Gigabit Ethernet, Fast Gigabit Ethernet, full duplex

The Ethernet model contains a MAC model (`EtherMAC`), LLC model (`EtherLLC`) as well as a bus (`EtherBus`, for modelling coaxial cable) and a hub (`EtherHub`) model. A switch model (`EtherSwitch`) is also provided.

- `EtherHost` is a sample node with an Ethernet NIC;
- `EtherSwitch`, `EtherBus`, `EtherHub` model switching hub, repeating hub and the old coxial cable;
- basic compnents of the model: `EtherMAC`, `EtherLLC`/`EtherEncap` module types, `MACRelayUnit` (`MACRelayUnitNP` and `MACRelayUnitPP`), `EtherFrame` message type, `MACAddress` class

Sample simulations:

- the `MixedLAN` model contains hosts, switch, hub and bus
- the `LargeNet` model contains hundreds of computers, switches and hubs (numbers depend on model configuration in largenet.ini) and mixes all kinds of Ethernet technologies

5.1.1 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

5.2 Physical layer

The nodes of the Ethernet networks are connected by coaxial, twisted pair or fibre cables. 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 `DatarateConnection` 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
- `10Gbps` Fast Gigabit Ethernet

5.2.1 EtherBus

The `EtherBus` component can model a common coaxial cable found in earlier Ethernet LANs. The nodes are attached at specific positions of the cable. If a node sends a packet, it is transmitted in both direction by a given propagation speed.

The gates of the `EtherBus` represent taps. The positions of the taps are given by the `positions` parameter as a space separated list of distances in metres. If there are more gates then positions given, the last distance is repeated. The bus component send the incoming message in one direction and a copy of the message to the other direction (except at the ends). The propagation delays are computed from the distances of the taps and the `propagationSpeed` parameter.

Messages are not interpreted by the bus model in any way, thus the bus model is not specific to Ethernet in any way. Messages may represent anything, from the beginning of a frame transmission to end (or abortion) of transmission.

5.2.2 EtherHub

Ethernet hubs are a 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.

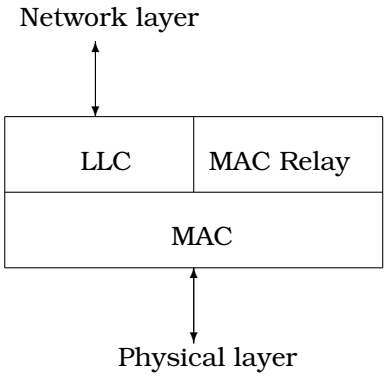
Messages are not interpreted by the `EtherType` hub model in any way, thus the hub model is not specific to Ethernet. Messages may represent anything, from the beginning of a frame transmission to end (or abortion) of transmission.

The hub module collects the following statistics:

- `pkBytes` handled packets length (vector)
- `messages/sec` number of packets per seconds (scalar)

5.3 MAC layer

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 does not do encapsulation/decapsulation), it is implemented in a separate modules (`EtherEncap` and `EtherLLC`) 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 behaviour of the MAC component is much simpler than in classic Ethernet that used coaxial cables and hubs. The INET framework contains two MAC modules for Ethernet: the `EtherMACFullDuplex` is simpler to understand and easier to extend, because it supports only full-duplex connections. The `EtherMAC` module implements the full MAC functionality including CSMA/CD, it can operate both half-duplex and full-duplex mode.

Packets and frames

The environment of the MAC modules is described by the `IEtherMAC` module interface. Each MAC modules has gates to connect to the physical layer (`phys$i` and `phys$o`) and to connect to the upper layer (LLC module is hosts, relay units in switches): `upperLayerIn` and `upperLayerOut`.

When a frame is received from the higher layers, it must be an `EtherFrame`, and with all protocol fields filled out (including the destination MAC address). The source address, if left empty, will be filled in with the configured `address` of the MAC.

Packets received from the network are `EtherTraffic` objects. They are messages representing inter-frame gaps (`EtherPadding`), jam signals (`EtherJam`), control frames (`EtherPauseFrame`) or data frames (all derived from `EtherFrame`). Data frames are passed up to the higher layers without modification. In `promiscuous` mode, the MAC passes up all received frames; otherwise, only the frames with matching MAC addresses and the broadcast frames are passed up.

Also, the module properly responds to PAUSE frames, but never sends them by itself – however, it transmits PAUSE frames received from upper layers. See section 5.3 for more info.

Queueing

When the transmission line is busy, messages received from the upper layer needs to be queued.

In routers, MAC relies on an external queue module (see `OutputQueue`), and requests packets from this external queue one-by-one. The name of the external queue must be given as the `queueModule` parameeer. There are implementations of `OutputQueue` to model finite buffer, QoS and/or RED.

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

PAUSE handling

The 802.3x standard supports PAUSE frames as a means of flow control. The frame contains a timer value, expressed as a multiple of 512 bit-times, that specifies how long the transmitter should remain quiet. If the receiver becomes uncongested before the transmitter's pause timer expires, the receiver may elect to send another PAUSE frame to the transmitter with a timer value of zero, allowing the transmitter to resume immediately.

`EtherMAC` will properly respond to PAUSE frames it receives (`EtherPauseFrame` class), however it will never send a PAUSE frame by itself. (For one thing, it doesn't have an input buffer that can overflow.)

`EtherMAC`, however, transmits PAUSE frames received by higher layers, and `EtherLLC` can be instructed by a command to send a PAUSE frame to MAC.

Error handling

If the MAC is not connected to the network ("cable unplugged"), it will start up in "disabled" mode. A disabled MAC simply discards any messages it receives. It is currently not supported to dynamically connect/disconnect a MAC.

CRC checks are modeled by the `bitError` flag of the packets. Erronous packets are dropped by the MAC.

Signals and statistics

Both MAC modules emits the following signals:

- `txPk` after successful data frame transmission, the data frame
- `rxPkOk` after successful data frame reception, the data frame
- `txPausePkUnits` after PAUSE frame sent, the pause time
- `rxPausePkUnits` after PAUSE frame received, the pause time
- `rxPkFromHL` when a data frame received from higher layer, the data frame
- `dropPkNotForUs` when a data frame received not addressed to the MAC, the data frame

- `dropPkBitError` when a frame received with bit error, the frame
- `dropPkIfaceDown` when a message received and the MAC is not connected, the dropped message
- `packetSentToLower` before starting to send a packet on phys$o gate, the packet
- `packetReceivedFromLower` after a packet received on phys$i gate, the packet (excluding PAUSE and JAM messages and dropped data frames)
- `packetSentToUpper` before sending a packet on upperLayerOut, the packet
- `packetReceivedFromUpper` after a packet received on upperLayerIn, the packet

Apart from statistics can be generated from the signals, the modules collects the following scalars:

- `simulated time` total simulation time
- `full duplex` boolean value, indicating whether the module operated in full-duplex mode
- `frames/sec sent` data frames sent (not including PAUSE frames) per second
- `frames/sec rcvd` data frames received (not including PAUSE frames) per second
- `bits/sec sent`
- `bits/sec rcvd`

Note that four of these scalars could be recorded as the count and value of the `txPkBytesSignal` and `rxPkBytesSignal` signals resp.

Visual effects

In the graphical environment, some animation effects help to follow the simulation. The color of the transmission channel is changed to yellow during transmission, and turns to red when collision detected. The icon of disconnected MAC modules are grayed out.

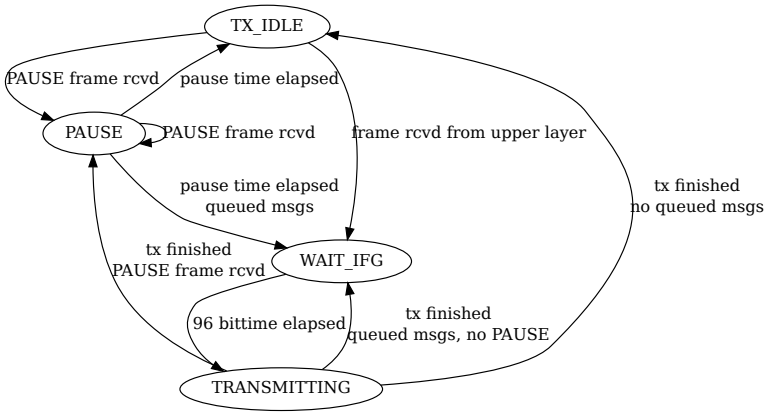
The icon of the Ethernet NICs are also colored according to the state of MAC module: yellow if transmitting, blue if receiving, red if collision detected, white in backoff and gray in paused state.

5.3.1 EtherMACFullDuplex

From the two MAC implementation `EtherMACFullDuplex` 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. Flow control works as described in section 5.3.

In the `EtherMACFullDuplex` module, packets arrived at the `phys$i` gate are handled when their last bit received.

Outgoing packets are transmitted according to the following state diagram:



The `EtherMACFullDuplex` module records two scalars in addition to the ones mentioned earlier:

- `rx channel idle (%)`: reception channel idle time as a percentage of the total simulation time

- `rx channel utilization (%)`: total reception time as a percentage of the total simulation time

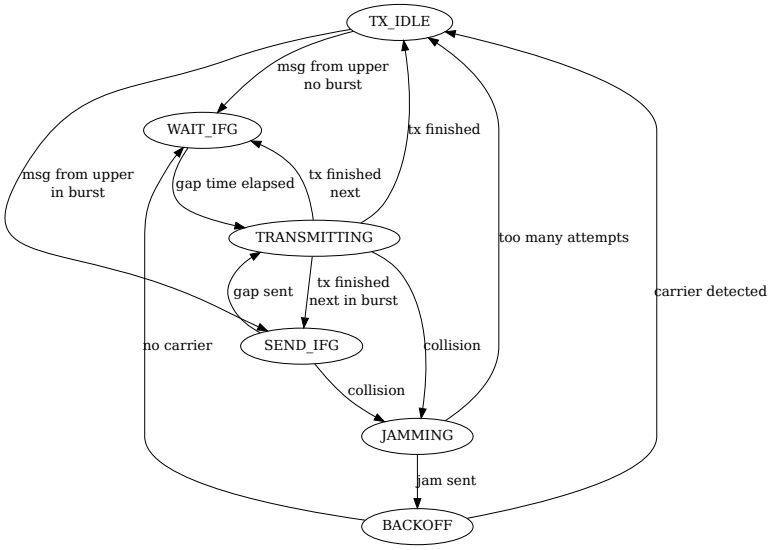
5.3.2 EtherMAC

Ethernet MAC layer implementing CSMA/CD. It supports both half-duplex and full-duplex operations; in full-duplex mode it behaves as `EtherMACFullDuplex`. In half-duplex mode it detects collisions, sends jam messages and retransmit 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`.

Unlike `EtherMACFullDuplex`, this MAC module processes the incoming packets when their first bit is received. The end of the reception is calculated by the MAC and detected by scheduling a self message.

When frames collide the transmission is aborted – in this case the transmitting station transmits a jam signal. Jam signals are represented by a `EtherJam` message. The jam message contains the tree identifier of the frame whose transmission is aborted. When the `EtherMAC` receives a jam signal, it knows that the corresponding transmission ended in jamming and have been aborted. Thus when it receives as many jams as collided frames, it can be sure that the channel is free again. (Receiving a jam message marks the beginning of the jam signal, so actually has to wait for the duration of the jamming.)

The operation of the MAC module can be schematized by the following state chart:



The module generates these extra signals:

- `collision` when collision starts (received a frame, while transmitting or receiving another one; or start to transmit while receiving a frame), the constant value 1
- `backoff` when jamming period ended and before waiting according to the exponential backoff algorith, the constant value 1

These scalar statistics are generated about the state of the line:

- `rx channel idle (%)` reception channel idle time (full duplex) or channel idle time (half-duplex), as a percentage of the total simulation time
- `rx channel utilization (%)` total successful reception time (full-duplex) or total successful reception/transmission time (half duplex), as a percentage of the total simulation time
- `rx channel collision (%)` total unsuccessful reception time, as a percentage of the total simulation time
- `collisions` total number collisions (same as count of `collisionSignal`)
- `backoffs` total number of backoffs (same as count of `backoffSignal`)

5.4 Switches

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 routes data frames between the connected subnets.

While a hub repeats the data frames on each connected line, possibly causing collisions, switches help to segment the network to small collision domains. In modern Gigabit LANs each node is connected to the switch direclty 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.

5.4.1 MAC relay units

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 `IEtherMAC` modules.

Internally the relay unit holds a table for the destination address -> output port mapping. When it receives a data frame it updates the table with the source address->input port. The table can also be pre-loaded from a text file while initializing the relay unit. The file name given as the `addressTableFile` parameter. Each line of the file contains a hexadecimal MAC address and a decimal port number separated by tabs. Comment lines beginning with '#' are also allowed:

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

The size of the lookup table is restricted by the `addressTableSize` parameter. When the table is full, the oldest address is deleted. Entries are also deleted if their age exceeds the duration given as the `agingTime` parameter.

If the destination address is not found in the table, the frame is broadcasted. The frame is not sent to the same subnet it was received from, because the target already received the original frame. The only exception if the frame arrived through a radio channel, in this case the target can be out of range. The port range 0..`numWirelessPorts`-1 are reserved for wireless connections.

The `IMACRelayUnit` module is not a concrete implementation, it just defines gates and parameters an `IMACRelayUnit` should have. Concrete inplementations add capacity and performance aspects to the model (number of frames processed per second, amount of memory available in the switch, etc.) C++ implementations can subclass from the class `MACRelayUnitBase`.

There are two versions of `IMACRelayUnit`:

`MACRelayUnitNP` models one or more CPUs with shared memory, working from a single shared queue.

`MACRelayUnitPP` models one CPU assigned to each incoming port, working with shared memory but separate queues.

In both models input messages are queued. CPUs poll messages from the queue and process them in `processingTime`. If the memory usage exceeds `bufferSize`, the frame will be dropped.

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.

The relay units collects the following statistics:

usedBufferBytes memory usage as function of time

processedBytes count and length of processed frames

droppedBytes count and length of frames dropped caused by out of memory

5.4.2 EtherSwitch

Model of an Ethernet switch containing a relay unit and multiple MAC units.

The duplexChannel attributes 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 half duples MAC with CSMA/CD.

NOTE: Switches don't implement the Spanning Tree Protocol. You need to avoid cycles in the LAN topology.

5.5 Link Layer Control

5.5.1 Frame types

The raw 802.3 frame format header contains the MAC addresses of the destination and source of the packet and the length of the data field. The frame footer contains the FCS (Frame Check Sequence) field which is a 32-bit CRC.

MAC destination	MAC source	Length	Payload	FCS
6 octets	6 octets	2 octets	46-1500 octets	4 octets

Each such frame is preceded by a 7 octet Preamble (with 10101010 octets) and a 1 octet SFD (Start of Frame Delimiter) field (10101011) and followed by an 12 octet interframe gap. These fields are added and removed in the MAC layer, so they are omitted here.

When multiple upper layer protocols use the same Ethernet line, the kernel has to know which which component handles the incoming frames. For this purpose a protocol identifier was added to the standard Ethernet frames.

The first solution preceded the 802.3 standard and used a 2 byte protocol identifier in place of the Length field. This is called Ethernet II or DIX frame. Each protocol id is above 1536, so the Ethernet II frames and the 802.3 frames can be distinguished.

MAC destination	MAC source	EtherType	Payload	FCS
6 octets	6 octets	2 octets	46-1500 octets	4 octets

The LLC frame format uses a 1 byte source, a 1 byte destination, and a 1 byte control information to identify the encapsulated protocol adopted from the 802.2 standard. These fields follow the standard 802.3 header, so the maximum length of the payload is 1497 bytes:

MAC destination	MAC source	Length	DSAP	SSAP	Control	Payload	FCS
6 octets	6 octets	2 octets	1 octets	1 octets	1 octets	43-1497 octets	4 octets

The SNAP header uses the EtherType protocol identifiers inside an LLC header. The SSAP and DSAP fields are filled with 0xAA (SAP_SNAP), and the control field is 0x03. They are followed by a 3 byte orgnaization and a 2 byte local code the identify the protocol. If the organization code is 0, the local field contains an EtherType protocol identifier.

MAC destination	MAC source	Length	DSAP 0xAA	SSAP 0xAA	Control 0x03	OrgCode	Local Code	Payload	FCS
6 octets	6 octets	2 octets	1 octets	1 octets	1 octets	3 octets	2 octets	38-1492 octets	4 octets

The INET defines these frames in the `EtherFrame.msg` file. The models supports Ethernet II, 803.2 with LLC header, and 803.3 with LLC and SNAP headers. The corresponding classes are: `EthernetIIFrame`, `EtherFrameWithLLC` and `EtherFrameWithSNAP`. They all class from `EtherFrame` which only represents the basic MAC frame with source and destination addresses. `EtherMAC` only deals with `EtherFrame`s, and does not care about the specific subclass.

Ethernet frames carry data packets as encapsulated cMessage objects. Data packets can be of any message type (cMessage or cMessage subclass).

The model encapsulates data packets in Ethernet frames using the `encapsulate()` method of cMessage. Encapsulate() updates the length of the Ethernet frame too, so the model doesn't have to take care of that.

The fields of the Ethernet header are passed in a `Ieee802Ctrl` control structure to the LLC by the network layer.

EtherJam, EtherPadding (interframe gap), EtherPauseFrame?

5.5.2 EtherEncap

The `EtherEncap` module generates `EthernetIIFrame` messages.

EtherFrameII

5.5.3 EtherLLC

EtherFrameWithLLC

SAP registration

`EtherLLC` and higher layers

The `EtherLLC` module can serve several applications (higher layer protocols), and dispatch data to them. Higher layers are identified by DSAP. See section "Application registration" for more info.

`EtherEncap` doesn't have the functionality to dispatch to different higher layers because in practice it'll always be used with IP.

Communication between LLC and Higher Layers

Higher layers (applications or protocols) talk to the `EtherLLC` module.

When a higher layer wants to send a packet via Ethernet, it just passes the data packet (a cMessage or any subclass) to `EtherLLC`. The message kind has to be set to IEEE802CTRL_DATA.

In general, if `EtherLLC` receives a packet from the higher layers, it interprets the message kind as a command. The commands include IEEE802CTRL_DATA (send a frame), IEEE802CTRL_REGISTER_DSAP (register highher layer) IEEE802CTRL_DEREGISTER_DSAP (deregister higher layer) and IEEE802CTRL_SENDPAUSE (send PAUSE frame) – see EtherLLC for a more complete list.

The arguments to the command are NOT inside the data packet but in a "control info" data structure of class `Ieee802Ctrl`, attached to the packet. See controlInfo() method of cMessage (OMNeT++ 3.0).

For example, to send a packet to a given MAC address and protocol identifier, the application sets the data packet's message kind to ETH_DATA ("please send this data packet" command), fills in the `Ieee802Ctrl` structure with the destination MAC address and the protocol identifier, adds the control info to the message, then sends the packet to `EtherLLC`.

When the command doesn't involve a data packet (e.g. IEEE802CTRL_(DE)REGISTER_DSAP, IEEE802CTRL_SENDPAUSE), a dummy packet (empty cMessage) is used.

Rationale

The alternative of the above communications would be:

- adding the parameters such as destination address into the data packet. This would be a poor solution since it would make the higher layers specific to the Ethernet model.

- encapsulating a data packet into an *interface packet* which contains the destination address and other parameters. The disadvantages of this approach is the overhead associated with creating and destroying the interface packets.

Using a control structure is more efficient than the interface packet approach, because the control structure can be created once inside the higher layer and be reused for every packet.

It may also appear to be more intuitive in Tkenv because one can observe data packets travelling between the higher layer and Ethernet modules – as opposed to "interface" packets.

EtherLLC: SAP Registration

The Ethernet model supports multiple applications or higher layer protocols.

So that data arriving from the network can be dispatched to the correct applications (higher layer protocols), applications have to register themselves in `EtherLLC`. The registration is done with the IEEE802CTRL_REGISTER_DSAP command (see section "Communication between LLC and higher layers") which associates a SAP with the LLC port. Different applications have to connect to different ports of `EtherLLC`.

The ETHERCTRL_REGISTER_DSAP/IEEE802CTRL_DEREGISTER_DSAP commands use only the dsap field in the `Ieee802Ctrl` structure.

5.5.4 EthernetInterface module

The EthernetInterface compound module implements the IWiredNic interface. Complements EtherMAC and EtherEncap with an output queue for QoS and RED support. It also has configurable input/output filters as IHook components similarly to the PPPInterface module.

5.6 Ethernet applications

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

The server component of the model (EtherAppSrv) responds with a 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 EtherLLC module which requires registration of the SAP before sending frames.

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

The client and server application works 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 EtherLLC or an EthernetInterface NIC module.

The model also contains a host component that groups the applications and the LLC and MAC components together (EtherHost). 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).

5.7 Ethernet networks

5.7.1 LargeNet model

The LargeNet model demonstrates how one can put together models of large LANs with little effort, making use of MAC auto-configuration.

LargeNet models a large Ethernet campus backbone. As configured in the default omnetpp.ini, it contains altogether about 8000 computers and 900 switches and hubs. This results in about 165MB process size on my (32-bit) linux box when I run the simulation. The model mixes all kinds of Ethernet technology: Gigabit Ethernet, 100Mb full duplex, 100Mb half duplex, 10Mb UTP, 10Mb bus ("thin Ethernet"), switched hubs, repeating hubs.

The topology is in LargeNet.ned, and it looks like this: there's chain of n=15 large "backbone" switches (switchBB[]) as well as four more large switches (switchA, switchB, switchC, switchD) connected to somewhere the middle of the backbone (switchBB[4]). These 15+4 switches make up the backbone; the n=15 number is configurable in omnetpp.ini.

Then there're several smaller LANs hanging off each backbone switch. There're three types of LANs: small, medium and large (represented by compound module types SmallLAN, MediumLAN, LargeLAN). A small LAN consists of a few computers on a hub (100Mb half duplex);

a medium LAN consists of a smaller switch with a hub on one of its port (and computers on both); the large one also has a switch and a hub, plus an Ethernet bus hanging of one port of the hub (there's still hubs around with one BNC connector besides the UTP ones). By default there're 5..15 LANs of each type hanging off each backbone switch. (These numbers are also omnetpp.ini parameters like the length of the backbone.)

The application model which generates load on the simulated LAN is simple yet powerful. It can be used as a rough model for any request-response based protocol such as SMB/CIFS (the Windows file sharing protocol), HTTP, or a database client-server protocol.

Every computer runs a client application (`EtherAppCli`) which connects to one of the servers. There's one server attached to switches A, B, C and D each: serverA, serverB, serverC and serverD – server selection is configured in omnetpp.ini). The servers run `EtherAppSrv`. Clients periodically send a request to the server, and the request packet contains how many bytes the client wants the server to send back (this can mean one or more Ethernet frames, depending on the byte count). Currently the request and reply lengths are configured in omnetpp.ini as intuniform(50,1400) and truncnormal(5000,5000).

The volume of the traffic can most easily be controlled with the time period between sending requests; this is currently set in omnetpp.ini to exponential(0.50) (that is, average 2 requests per second). This already causes frames to be dropped in some of the backbone switches, so the network is a bit overloaded with the current settings.

The model generates extensive statistics. All MACs (and most other modules too) write statistics into omnetpp.sca at the end of the simulation: number of frames sent, received, dropped, etc. These are only basic statistics, however it still makes the scalar file to be several ten megabytes in size. You can use the analysis tools provided with OMNeT++ to visualized the data in this file. (If the file size is too big, writing statistics can be disabled, by putting **.record-scalar=false in the ini file.) The model can also record output vectors, but this is currently disabled in omnetpp.ini because the generated file can easily reach gigabyte sizes.

Chapter 6

The Physical Environment

6.1 Overview

Todays wireless communication networks are heavily affected by their physical environment. Mobile networks operate in densely built urban areas, wireless networks work inside large buildings, low power wireless sensors communicate in industrial environments, batteries operate in various external conditions, and so on.

The propagation of communication signals, the movement of communicating agents, or battery exhaustion depend on the surrounding physical environment. For example, signals can be absorbed by objects, can pass through objects, can be refracted by surfaces, can be reflected from surfaces, etc. These effects cannot be ignored in high fidelity simulations.

The physical environment model is separated from the rest of the communication related models. It's main goal is to describe buildings, walls, vegetation, terrain, weather and other physical conditions that might have effect on signal propagation, movement, batteries, etc. The separation makes the model reusable by other simulation models that depend on these circumstances.

This chapter provides a brief overview of the physical environment model.

6.2 Physical Objects

The most important aspect of the physical environment is the objects which are present in it. For example, simulating an indoor wifi scenario needs to model walls, floors, ceilings, doors, windows, furniture, etc. Most of these objects have very basic shapes and homogeneous materials. So the physical object model has the following properties:

- `shape`: describes the 3D shape of the object independently of its position and orientation
- `position`: determines where the object is located in the 3D space
- `orientation`: determines how the object is rotated relative to its default orientation
- `material`: describes various material specific physical properties
- `graphical properties`: provides parameters for better visualization

The physical objects are also mostly immobile. In the current model 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. Concave shapes are not yet supported, such shapes should be split up into convex parts. The current implementation provides the following convex shapes:

- `Sphere`: specified by a radius
- `Cuboid`: specified by a length, a width, and a height
- `Prism`: specified by a 2 dimensional polygon base and a height
- `Polyhedron`: specified by a set of 3 dimensional vertices

In order to model the physical environment in detail a scenario might contain several thousands or even more physical objects. Other simulation models might need to query these objects quite often. For example, when the physical layer computes obstacle loss for a transmission, it needs to find the obstructing physical objects for each receiver. This requires computing the intersection between physical objects and the path traveled by the signal. For this purpose, the physical environment supports storing physical objects in one of the following efficient cache data structures:

- `GridObjectCache`: organizes objects into a spatial grid with a configurable constant cell size, cells contain those objects that intersect with them
- `BVHObjectCache`: organizes objects into a tree data structure, tree leaves contain a configurable number of closely positioned objects

6.3 Initialization

In order to easily define thousands of physical objects the model needs to use a flexible and concise representation. For this purpose, the physical environment provides an XML file format. This file can be used to define reusable shapes, reusable materials, and physical objects. The following example demonstrates the XML file format:

```
<environment>
  <!-- defines a sphere shape -->
  <shape id="1" type="sphere" radius="10"/>
  <!-- defines a cuboid shape -->
  <shape id="2" type="cuboid" size="20 30 40"/>
  <!-- defines a prism (e.g. a cube) shape -->
  <shape id="3" type="prism" height="100"
         points="0 0 100 0 100 100 0 100"/>
  <!-- defines a polyhedron (e.g. a cube) shape -->
  <shape id="4" type="polyhedron"
         points="0 0 0 100 0 0 100 100 0 0 100 0 ..."/>
  <!-- defines a material -->
  <material id="1" resistivity="100"
            relativePermittivity="4.5" relativePermeability="1"/>
  <!-- defines an object using the previously defined shapes and materials -->
  <object position="min 100 200 0" orientation="45 -30 0"
```

```
          shape="1" material="1"
          line-color="0 0 0" fill-color="112 128 144" opacity="0.5"/>
  <!-- defines another object similar to the previous one
       using an inline shape and a predefined material -->
  <object position="min 100 200 0" orientation="45 -30 0"
          shape="cuboid 20 30 40" material="concrete"
          line-color="0 0 0" fill-color="112 128 144" tags="Building"/>
</environment>
```

For more details on the XML file format please refer to the documentation in the `PhysicalEnvironment` NED file.

6.4 Visualization

Understanding the abstract physical environment model from numerical log output or debugging other simulation models based on the numerical data structures is difficult. Therefore the physical environment supports visualizing the physical objects on the graphical user interface. The visualization is done by drawing the physical objects on the parent compound module as a separate layer. In order to help distinguish among physical objects they have the following graphical properties:

- `line width`: surface outline
- `line color`: surface outline
- `fill color`: surface filling
- `opacity`: surface outline and filling
- `tags`: allows filtering objects on the graphical user interface

Although the physical objects are really modeled in 3 dimensions, the visualization is only 2 dimensional. The projection which converts physical objects to 2 dimensional shapes can be parameterized with an arbitrary view angle. The default view angle is the Z axis, in which case the layer shows the physical objects from above. The view angle can be changed during runtime using the parameter editor of the graphical user interface.

The projection is also used by other models for their own visualizations. This approach makes visualizations well integrated with each other. For example, the mobility models position network nodes, the physical layer draws ongoing transmissions, the obstacle loss draws intersections, etc. on the canvas according to the projection described above.

Chapter 7

The Power Model

7.1 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. The high fidelity simulation of power consumption allows designing power sensitive routing protocols, MAC protocols, physical layers, etc. which in turn results in more energy efficient devices.

In order to help the modeling process the power model is separated from the other simulation models. This separation makes the 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

This chapter provides a brief overview of the power model.

7.2 Energy Consumer Models

The energy consumer models describe the energy consumption of devices over time. For example, a radio consumes energy when it transmits or receives signals, or a CPU consumes energy when the network layer processes packets, or a display consumes energy when it's turned on, etc. Energy consumers connect to an energy storage that provides the consumed energy.

The `StateBasedEnergyConsumer` module implements a simple radio energy consumption model in the physical layer. This model determines the current power consumption of the radio using constant module parameters for each valid combination of the radio mode, the transmitter state and the receiver state.

7.3 Energy Generator Models

The energy generator models describe the energy generation of devices over time. A solar panel, for example, produces energy based on time, the panel's position 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.

The `AlternatingEnergyGenerator` module implements a simple model which alternates between two modes called generation mode and sleep mode. In generation mode it generates a randomly selected constant power for a random time interval. In sleep mode it doesn't generate energy for another random time interval.

7.4 Energy Storage Models

The energy storage models describe devices that absorb energy produced by generators, and provide energy for consumers. For example, an electrochemical battery in a mobile phone provides energy for its display, its CPU, and its wireless communication devices. It can also absorb energy produced by a solar panel installed on its display, or by a portable charger plugged into the wall socket.

The `SimpleEnergyStorage` implements an energy storage model that maintains a residual capacity by integrating the difference between the total generated power and the total consumed power over time. It can initiate node shutdown when the residual capacity drops below a configured threshold. It can also initiate node start when the residual capacity raises above another configured threshold. This simple model doesn't have properties such a memory effect, self-discharge, overcharging, temperature-dependence, etc. that real world batteries have.

Chapter 8

The Physical Layer

8.1 Overview

TODO: motivation in general? radio frames shared transmission, immutable data structures?

In the field of communication system simulations one of the most time consuming task is to simulate the physical layer. The simulation of transmissions, receptions and interferences in detail may often result in unacceptable performance. Finding the right abstractions, the right level of detail is difficult and very important.

The physical layer in INET was designed with the following goals in mind:

- customizability
- extensibility
- scalable level of detail
- ability to exploit parallel hardware

This chapter provides a brief overview of the physical layer model. For more details on the available modules, their parameterization and the actual implementations please refer to the documentation in the corresponding NED and C++ source files.

8.1.1 Customizability

The physical layer provides a wide variety of parameters to control its behavior. The most common parameters are physical quantities with physical units such as transmission power `[W]`, reception sensitivity `[W]`, carrier frequency `[Hz]`, propagation speed `[m/s]`, SNIR threshold `[dB]`, communication range `[m]`, bitrate `[b/s]`, etc.

Another commonly used parameter kind is the one that selects among alternative implementations by providing its name. Different implementations are separate modules that come with their own set of parameters to avoid the confusion of mixing unrelated parameters. Some modules may be split into more submodules further deepening the module hierarchy.

8.1.2 Extensibility

The physical layer is designed to be extensible with alternative implementations for various parts of the model. This is realized by separately defining C++ and NED interfaces between modules, and also by providing parameters in their parent modules to easily select among the available implementations.

New models can be added by implementing the necessary interfaces with optionally deriving from already existing ones. This allows the user to implement new models with less effort and focus on the real differences while the rest of the physical layer model remains the same.

8.1.3 Scalable Level of Detail

The physical layer is designed to be scalable with respect to the simulated level of detail. There are many possible ways to model various physical layer aspects. The main difference lies in the trade-off between performance versus accuracy.

The model might vary from simple statistical to accurate emulation. The simplest models ignore actual bits or even signal power for that matter. The most accurate models use bit precise signal representation and emulate most functions of real hardware in detail.

The software architecture might vary from flat to layered. A flat architecture is efficient but not modular. Functionality can only be affected through simple parameters and not by providing alternative implementations. Whereas a layered architecture is more flexible at the cost of more complex data structures, more data conversions, more resource management, and thus slower processing. On the other hand, it provides more customization opportunities to replace parts with alternative implementations and do research in this area.

The data representation might vary from scalar to multidimensional values. In the physical layer data quite often changes over time, frequency, space, or any combination thereof. The most obvious example is the analog signal power but there are others such as signal phase, bitrate or the signal to noise ratio.

The number of messages added to the future event queue might vary from one to the number of radios per transmission. One message might be sufficient for example, if the transmission is intended to a single destination and other receivers are either not affected or the effect is negligible with respect to accuracy. On the other hand, sometimes it might be necessary to process all transmissions by all receivers in order to have the desired effect on higher layers.

8.1.4 Exploiting Parallel Hardware

The physical layer is designed to utilize parallel hardware, multi-core CPUs, vector instructions and the highly parallel GPU. The computation of transmission arrival space-time coordinates, receptions, interferences, reception indications provide a good parallelization opportunity, because they dominate the physical layer performance and are independent of each other.

The medium module is a central component in the software architecture where parallel computation can happen. One possibility is optimistic parallel computation of results in multiple background threads while the main simulation thread continues normal execution. As new transmissions enter the channel the affected and already computed results are invalidated and the affected ongoing optimistic parallel computations are canceled.

8.2 The Radio Module

The radio module represents the physical device that is capable of transmitting and receiving signals on the medium. It consists of an antenna, a transmitter, a receiver and a power consumer model. It supports the following externally configurable radio modes:

- `off`: communication isn't possible and power consumption is zero
- `sleep`: communication isn't possible and power consumption is minimal
- `receiver`: only reception is possible and power consumption is low
- `transmitter`: only transmission is possible and power consumption is high
- `transceiver`: reception and transmission is simultaneously possible and power consumption is high
- `switching`: communication isn't possible and power consumption is minimal

The radio module also supports non-zero time radio mode switching. In addition, the transmitter and the receiver parts have separate states which describe what they are doing at the moment. Changes to these states are automatically published by the radio.

When the radio wants to transmit a signal on the medium it sends direct messages to other radios with the help of the central medium module. Receiver radios also handle the incoming messages with the help by the central medium module.

The radio module utilizes multiple submodules to further split its task. This design makes it extensible and customizable. The following sections describe the parts of the radio.

8.2.1 Antenna Models

The antenna models represent the physical device which converts electric signals into radio waves, and vice versa. They provide position and orientation using a mobility model that defaults to the mobility of the radio. They also compute the antenna gain based on antenna characteristics and 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 a linear interpolation according to a table index by the direction angles

The antenna models are in the `src/physicallayer/antenna/` directory.

8.2.2 Transmitter Models

This module represents the physical device which converts packets into electric signals. It takes a MAC frame and produces a description of the signal that is transmitted on the medium.

TODO

8.2.3 Receiver Models

This module represents the physical device which converts electric signals into packets. It takes a signal along with the interference computed by the medium and it produces a MAC frame along with a reception indication.

TODO

8.2.4 Error Models

TODO

8.2.5 Power Consumer Models

TODO: hivatkozas a power model-re?

The power consumer models describe how the radio consumes power depending on the operational mode, the transmitted and the received signals. The following list provides some examples:

- `SimplePowerConsumer`: power consumption is determined by the radio mode, the transmitter state and the receiver state

The power consumer models are in the `src/physicallayer/power/` directory.

8.3 The Medium Module

The medium module represents the shared physical medium where communication takes place. It keeps track of radios, noise sources, ongoing transmissions, background and other noises. The medium computes when, where and how transmissions and noises arrive at receivers. It also efficiently provides the set of interfering transmissions and noises for the receiver to compute the reception indication. It doesn't send or handle messages on its own, but it rather acts as a mediator between radios.

The medium module utilizes multiple submodules to further split its task. This design makes it extensible and customizable. The following sections describe the parts of the medium.

8.3.1 Propagation Models

TODO: difficulty, motivation? with respect to transmitter and receiver mobility

The propagation models are separate modules which describe how signals propagate through space over time. They compute the arrival space-time coordinates for transmissions. The following list provides some examples:

- `ConstantTimePropagation`: propagation time is independent of the traveled distance and it's determined by a constant parameter
- `ConstantSpeedPropagation`: propagation time is proportional to the traveled distance determined by a constant propagation speed parameter

- `ConstantSpeedGPUPropagation`: propagation time is computed in parallel on the GPU for all receivers

The propagation models are in the `src/physicallayer/propagation/` directory.

8.3.2 Path Loss Models

TODO: difficulty, motivation?

The path loss models are separate modules which describe the reduction of power as the signal propagates through space. They compute the power loss factor based on the traveled distance, the signal frequency and the propagation speed. They may also provide the opposite, that is the traveled distance based on the power loss factor, the signal frequency and the propagation speed. The following list provides some examples:

- `FreeSpacePathLoss`
- `LogNormalShadowing`
- `TwoRayGroundReflection`
- `BreakpointPathLoss`
- `NakagamiFading`
- `RayleighFading`
- `RicianFading`
- `SUIPathLoss`
- `UWBIRStochasticPathLoss`
- etc.

The path loss models are in the `src/physicallayer/pathloss/` directory.

8.3.3 Obstacle Loss Models

TODO: hivatkozas a physical environment modelre?

The obstacle loss models describe the reduction of power as the signal propagates through obstacles. They compute the power loss factor based on the traveled straight path, the signal frequency and the obstructing physical objects. The following list provides some examples:

- `TracingObstacleLoss`: the power loss is based on accurate dielectric and reflection loss along the straight path considering the shape, position, orientation and material of physical objects

The obstacle loss models are in the `src/physicallayer/obstacleloss/` directory.

8.3.4 Background Noise Models

The background noise models describe how the background noise changes over space and time. They compute the noise signal for a given space-time interval.

- `IsotropicBackgroundNoise`: the noise is independent of time and position and its power is determined by a constant parameter

The background noise models are in the `src/physicallayer/backgroundnoise/` directory.

8.3.5 Neighbor Cache Models

The neighbor cache models provide an efficient way of keeping track of the neighbor relationship between radios. They maintain the potential set of receivers for all transmitters on the medium. They follow radio position changes but may provide conservative approximations. The following list provides some examples:

- `ListNeighborCache`: maintains a separate periodically updated neighbor list for each radio
- `GridNeighborCache`: organizes radios in a 3 dimensional grid with constant cell size and updates periodically
- `QuadTreeNeighborCache`: organizes radios in a 2 dimensional quad tree (ignoring the Z axis) with constant node size and updates periodically

The neighbor cache models are in the `src/physicallayer/neighborcache/` directory.

8.4 Signal Representations

TODO

8.4.1 Flat Representation

TODO

8.4.2 Layered Representation

TODO

Packet Domain

TODO

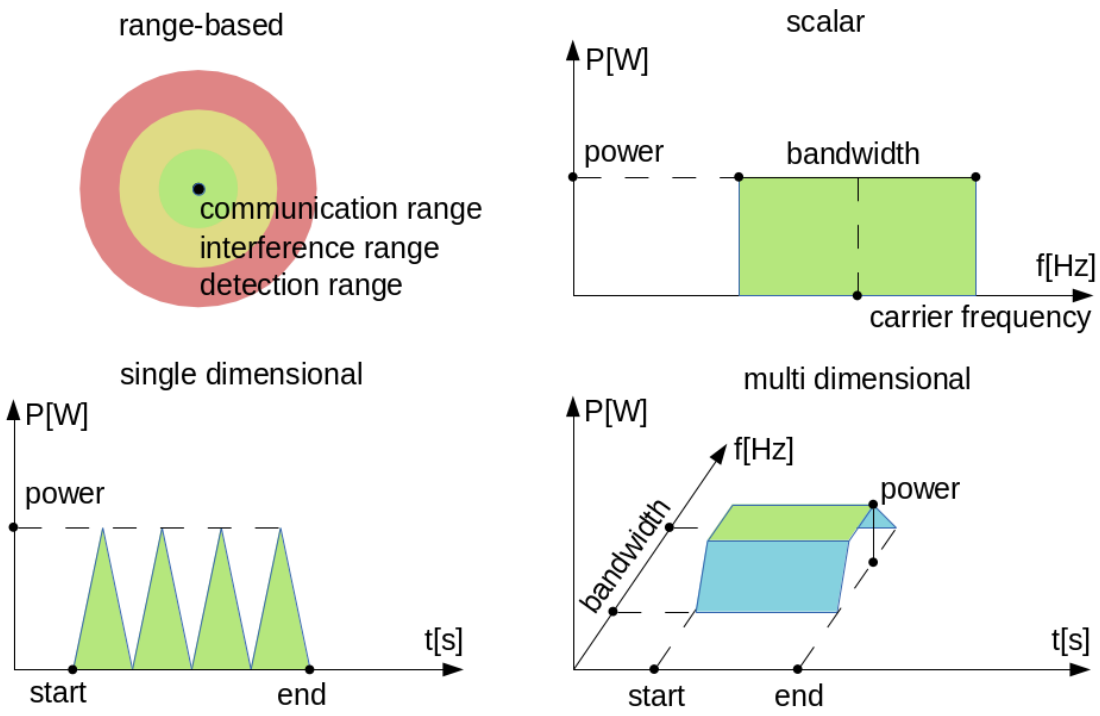


Figure 8.1: Analog signal representations

Bit Domain

TODO

Symbol Domain

TODO

Sample Domain

TODO

Analog Domain

TODO

8.5 Signal Processing

TODO

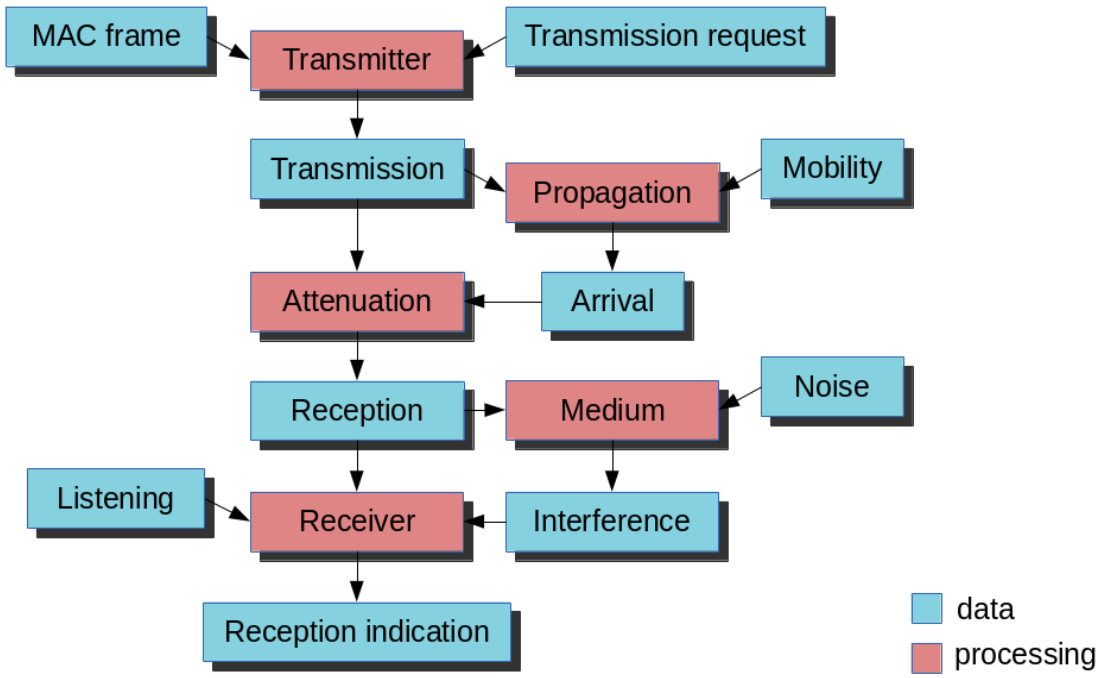


Figure 8.2: Data flow

The following sections describe the data structures of the data flow.

8.5.1 Transmission Request

This data structure contains parameters that control how the transmitter produces the transmission. It is attached as a control info object to the MAC frame sent down from the MAC module to the radio.

8.5.2 Transmission

This data structure represents the transmission of a radio signal. All models contain at least the start/end time, start/end antenna position, start/end antenna orientation of the transmitter.

8.5.3 Arrival

This data structure contains information about the space and time coordinates of a transmission arriving at a particular receiver. It contains at least the start/end time, start/end antenna position, start/end antenna orientation of the receiver.

8.5.4 Listening

TODO

8.5.5 Reception

This data structure represents the reception of a radio signal by a particular receiver. All models contain at least the start/end time, start/end antenna position, start/end antenna orientation of the receiver.

8.5.6 Interference

TODO

8.5.7 Reception Decision

TODO

8.5.8 Reception Indication

TODO

8.6 Visualization

The physical layer provides some parameters to display various internal state such as ongoing transmissions, recent successful receptions and recent obstacle intersections.

Chapter 9

The 802.11 Model

9.1 Overview

This chapter provides an overview of the IEEE 802.11 model for the INET Framework.

An IEEE 802.11 interface (NIC) comes in several flavours, differring in their role (ad-hoc station, infrastructure mode station, or access point) and their level of detail:

1. `Ieee80211Nic`: a generic (configurable) NIC
2. `Ieee80211NicAdhoc`: for ad-hoc mode
3. `Ieee80211NicAP`, `Ieee80211NicAPSimplified`: for use in an access point
4. `Ieee80211NicSTA`, `Ieee80211NicSTASimplified`: for use in an infrastructure-mode station

NICs consist of four layers, which are the following (in top-down order):

1. agent
2. management
3. MAC
4. physical layer (radio)

The physical layer modules (`Ieee80211Radio`; with some limitations, `SnrEval80211`, `Decider80211` can also be used) deal with modelling 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. The implementation of these modules is based on the Mobility Framework.

The MAC layer (`Ieee80211Mac`) performs transmission of frames according to the CSMA/CA protocol. It receives data and management frames from the upper layers, and transmits them.

The management layer performs encapsulation and decapsulation of data packets for the MAC, and 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 model the scan-authenticate-associate process, so they cannot be used in experiments involving handover.

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 layer is currenly only present in the `Ieee80211NicSTA` NIC module, as an `Ieee80211AgentSTA` module. The managament entities in other NIC variants do not have as much freedom as to need an agent to control them.

By modifying or replacing the agent, one can alter the dynamic behaviour of STAs in the network, for example implement different handover strategies.

9.1.1 Limitations

See the documentation of `Ieee80211Mac` for features unsupported by this model.

TODO further details about the implementation: what is modelled and what is not (beacons, auth, ...), communication between modules, frame formats, ...

Chapter 10

Node Mobility

10.1 Overview

In order to accurately evaluate a protocol for an ad-hoc network, it is important to use a realistic model for the motion of mobile hosts. Signal strengths, radio interference and channel occupancy depends on the distances between nodes. The choice of the mobility model can significantly influence the results of a simulation (e.g. data packet delivery ratio, end-to-end delay, average hop count) as shown in [CBD02].

There are two methods for incorporating mobility into simulations: using traces and synthetic models. Traces contains recorded motion of the mobile hosts, as observed in real life system. Synthetic models use mathematical models for describing the behaviour of the mobile hosts.

There are mobility models that represent mobile nodes whose movements are independent of each other (entity models) and mobility models that represent mobile nodes whose movements are dependent on each other (group models). Some of the most frequently used entity models are the Random Walk Mobility Model, Random Waypoint Mobility Model, Random Direction Mobility Model, Gauss-Markov Mobility Model, City Section Mobility Model. The group models include the Column Mobility Model, Nomadic Community Mobility Model, Pursue Mobility Model, Reference Point Group Mobility Model.

The INET framework has components for the following trace files:

- **Bonn Motion** native file format of the BonnMotion scenario generation tool.
- **Ns2** trace file generated by the CMU's scenario generator that used in Ns2.
- **ANSim** XML trace file of the ANSim (Ad-Hoc Network Simulation) tool.

It is easy to integrate new entity mobility models into the INET framework, but group mobility is not supported yet. Therefore all the models shipped with INET are implementations of entitiy models:

- **Deterministic Motions** for fixed position nodes and nodes moving on a linear, circular, rectangular paths.
- **Random Waypoint** model includes pause times between changes in destination and speed.

- **Gauss-Markov** model uses one tuning parameter to vary the degree of randomness in mobility pattern.
- **Mass Mobility** models a mass point with inertia and momentum.
- **Chiang Mobility** uses a probabilistic transition matrix to change the state of motion of the node.

10.2 Mobility in INET

In INET mobile nodes have to contain a module implementing the `IMobility` marker interface. This module stores the current coordinates of the node and is responsible for updating the position periodically and emitting the `mobilityStateChanged` signal when the position changed.

The p[0] and p[1] fields of the display string of the node is also updated, so if the simulation run is animated, the node is actually moving on the screen. The current position of the node can be obtained from the display string.

The radio simulations has a `ChannelControl` module that takes case of establishing communication channels between nodes that are within communication distance and tearing down the connections when they move out of range. The `ChannelControl` module uses to `mobilityStateChanged` signal to determine when the connection status needs to be updated.

There are two possibilities to implement a new mobility model. The simpler but limited one is to use `TurtleMobility` as the mobility component and to write a script similar to the turtle graphics of LOGO. The second is to implement a simple module in C++. In this case the C++ class of the mobility module should be derived from `IMobility` and its NED type should implement the `IMobility` interface.

10.2.1 MobilityBase class

The abstract `MobilityBase` class is the base of the mobility modules defined in the INET framework. This class allows to define a cubic volume that the node can not leave. The volume is configured by setting the `constraintAreaX`, `constraintAreaY`, `constraintAreaZ`, `constraintAreaWidth`, `constraintAreaHeight` and `constraintAreaDepth` parameters.

When the module is initialized it sets the initial position of the node by calling the `initializePosition()` method. The default implementation of this method sets the position from the display string if the `initFromDisplayString` parameter is true. Otherwise the position can be given as the `initialX`, `initialY` and `initialZ` parameters. If neither of these parameters are given, a random initial position is choosen within the contraint area.

The module is responsible for periodically updating the position. For this purpose it should send timer messages to itself. These messages are processed in the `handleSelfMessage` method. In derived classes, `handleSelfMessage` should compute the new position and update the display string and publish the new position by calling the `positionUpdated` method.

When the node reaches the boundary of the constraint area, the mobility component has to prevent the node to exit. It can call the `handleIfOutside` method, that offers the following policies:

- reflect of the wall
- reappear at the opposite edge (torus area)

- placed at a randomly chosen position of the area
- stop the simulation with an error

10.2.2 MovingMobilityBase

The abstract `MovingMobilityBase` class can be used to model mobilities when the node moves on a continous trajectory and updates its position periodically. Subclasses only need to implement the `move` method that is responsible to update the current position and speed of the node.

The abstract `move` method is called autmotically in every `updateInterval` steps. The method is also called when a client requested the current position or speed or when the `move` method requested an update at a future moment by setting the `nextChange` field. This can be used when the state of the motion changes at a specific time that is not a multiple of `updateInterval`. The method can set the `stationary` field to `true` to indicate that the node reached its final position and no more position update is needed.

10.2.3 LineSegmentsMobilityBase

The path of a mobile node often consist of linear movements of constant speed. The node moves with some speed for some time, then with another speed for another duration and so on. If a mobility model fits this description, it might be suitable to derive the implementing C++ class from `LineSegmentsMobilityBase`.

The module first choose a target position and a target time by calling the `setTargetPosition` method. If the target position differs from the current position, it starts to move toward the target and updates the position in the configured `updateInterval` intervals. When the target position reached, it chooses a new target.

10.3 Implemented models

10.3.1 Deterministic movements

StationaryMobility This mobility module does nothing; it can be used for stationary nodes.

StaticGridMobility Places all nodes in a rectangular grid.

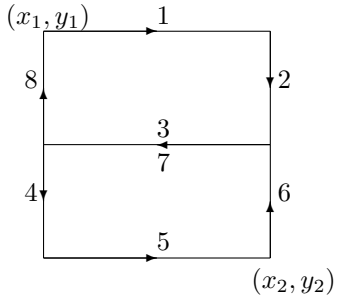
LinearMobility This is a linear mobility model with speed, angle and acceleration parameters. Angle only changes when the mobile node hits a wall: then it reflects off the wall at the same angle.

z coordinate is constant movement is always parallel with X-Y plane

CircleMobility Moves the node around a circle parallel to the X-Y plane with constant speed. The node bounces from the bounds of the constraint area. The circle is given by the `cx`, `cy` and `r` parameters, The initial position determined by the `startAngle` parameter. The position of the node is refreshed in `updateInterval` steps.

RectangleMobility Moves the node around the constraint area. configuration: speed, startPos, updateInterval

TractorMobility Moves a tractor through a field with a certain amount of rows. The following figure illustrates the movement of the tractor when the `rowCount` parameter is 2. The trajectory follows the segments in $1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3 \ldots$ order. The area is configured by the `x1`, `y1`, `x2`, `y2` parameters.



10.3.2 Random movements

RandomWPMobility In the Random Waypoint mobility model the nodes move in line segments. For each line segment, a random destination position (distributed uniformly over the playground) and a random speed is 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 the algorithm calculates a new random position, etc.

GaussMarkovMobility The Gauss-Markov model contains a tuning parameter, that control the randomness in the movement of the node. Let the magnitude and direction of speed of the node at the nth time step be s_n and d_n. The next speed and direction is computed as

$$s_{n+1} = \alpha s_n + (1-\alpha)\bar{s} + \sqrt{(1-\alpha^2)}s_{x_n}$$

$$d_{n+1} = \alpha s_n + (1-\alpha)\bar{d} + \sqrt{(1-\alpha^2)}d_{x_n}$$

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 Gaussian distribution.

Totally random walk (Brownian motion) is obtained by setting $\alpha = 0$, while $\alpha = 1$ results a 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 ($\bar{d}$) 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 a mass. It is the one used in [Pea99].

> "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 average of 5 seconds and standard deviation of 0.1 second. When it makes a turn, the

new direction (angle) in which it will move is a normally distributed random number with 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 a bit more, via the `changeInterval`, `changeAngleBy` and `changeSpeedBy` parameters. The parameters described above correspond to the following settings:

```
changeInterval = normal(5, 0.1)
changeAngleBy = normal(0, 30)
speed = normal(avgSpeed, 0.01)
```

ChiangMobility Chiang's random walk movement model ([Chi98]).

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:

$$\begin{pmatrix} 0 & 0.5 & 0.5 \\ 0.3 & 0.7 & 0 \\ 0.3 & 0 & 0.7 \end{pmatrix}$$

The `ChiangMobility` module supports the following parameters:

- `updateInterval` position update interval
- `stateTransitionInterval` state update interval
- `speed`: the speed of the node

ConstSpeedMobility `ConstSpeedMobility` does not use one of the standard mobility approaches. The user can define a velocity for each Host and an update interval. If the velocity is greater than zero (i.e. the Host is not stationary) the `ConstSpeedMobility` module calculates a random target position for the Host. Depending to the update interval and the velocity it calculates the number of steps to reach the destination and the step-size. Every update interval `ConstSpeedMobility` calculates the new position on its way to the target position and updates the display. Once the target position is reached `ConstSpeedMobility` calculates a new target position.

This component has been taken over from Mobility Framework 1.0a5.

10.3.3 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.

Ns2Mobility 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 parent module's index
- `scrollX`,`scrollY` user specified translation of the coordinates

ANSimMobility reads trace files of 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)>
<!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 to parent module's index

NOTE: The `ANSimMobility` module process only the `position_change` elements and it ignores the `start_time` attribute. It starts the move on the next segment immediately.

10.4 Mobility scripts

The `TurtleMobility` module can be parametrized by a script file containing LOGO-style movement commands in XML format.

The module has these parameters:

- `updateInterval` time interval to update the hosts position
- `constraintAreaX`, `constraintAreaY`, `constraintAreaWidth`, `constraintAreaHeight`: constraint area that the node can not leave
- `turtleScript` XML file describing the movements

The content of the XML file should conform to the following DTD (can be found as `TurtleMobility.dtd` in the source tree):

```
<!ELEMENT movements (movement)*>

<!ELEMENT movement (repeat|set|forward|turn|wait|moveto|moveby)*>
<!ATTLIST movement id NMTOKEN #IMPLIED>

<!ELEMENT repeat (repeat|set|forward|turn|wait|moveto|moveby)*>
<!ATTLIST repeat n CDATA #IMPLIED>

<!ELEMENT set EMPTY>
<!ATTLIST set x     CDATA #IMPLIED
              y     CDATA #IMPLIED
              speed CDATA #IMPLIED
              angle CDATA #IMPLIED
              borderPolicy (reflect|wrap|placerandomly|error) #IMPLIED>

<!ELEMENT forward EMPTY>
<!ATTLIST forward d CDATA #IMPLIED
                  t CDATA #IMPLIED>

<!ELEMENT turn EMPTY>
<!ATTLIST turn angle CDATA #REQUIRED>

<!ELEMENT wait EMPTY>
<!ATTLIST wait t CDATA #REQUIRED>

<!ELEMENT moveto EMPTY>
<!ATTLIST moveto x CDATA #IMPLIED
                 y CDATA #IMPLIED
                 t CDATA #IMPLIED>

<!ELEMENT moveby EMPTY>
<!ATTLIST moveby x CDATA #IMPLIED
                 y CDATA #IMPLIED
                 t CDATA #IMPLIED>
```

The file contains `movement` elements, each describing a trajectory. The `id` attribute of the `movement` element can be used to refer 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 state of motion is described by the following variables:

- `position`: (x, y) coordinate of the current location of the node
- `speed`, `angle`: magnitude and direction of the node's velocity
- `targetPos`: target position of the current line segment. If given the `speed` and `angle` is not used
- `targetTime` the end time of the current linear motion
- `borderPolicy`: one of
  - `reflect` the node reflects at the boundary,
  - `wrap` the node appears at the other side of the area,
  - `placerandomly` the node placed at a random position of the area,
  - `error` signals an error when the node reaches the boundary

The `movement` elements may contain the the following commands:

- `repeat(n)` repeats its content n times, or indefinetly if the n attribute is omitted.
- `set(x,y,speed,angle,borderPolicy)` modifies the state of the node.
- `forward(d,t)` moves the node for t time or to the d distance with the current speed. If both d and t is given, then the current speed is ignored.
- `turn(angle)` increase 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 meter per seconds. Attibutes 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 distibutions 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">
            <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 11

IPv4

11.1 Overview

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

The nodes that are connected to the Internet can be either a host or a router. The hosts can send and recieve 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 cards and perform packet routing between the connected networks. Routers does not need the transport layer, they work on the IP level only. The division between routers and hosts is not strict, because if a host have 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 an 1500 bytes 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.

11.1.1 INET modules

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 `RoutingTable` 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 to generate and process multicast group membership reports.

These modules are assembled into a complete network layer module called NetworkLayer. This module has dedicated gates for TCP, UDP, SCTP, RSVP, OSPF, Manet, and Ping higher layer protocols. It can be connected to several network interface cards: Ethernet, PPP, Wlan, or external interfaces. The NetworkLayer module is used to build IPv4 hosts (StandardHost) and routers (Router).

The implementation of these modules are based on the following RFCs:

- RFC791: Internet Protocol
- RFC792: Internet Control Message Protocol
- RFC826: Address Resolution Protocol
- RFC1122: Requirements for Internet Hosts - Communication Layers
- RFC2236: Internet Group Management Protocol, Version 2

The subsequent sections describe the IPv4 modules in detail.

11.2 The IPv4 Module

The IPv4 module implements the IPv4 protocol.

For connecting the upper layer protocols the IPv4 module has *transportIn[]* and *transportOut[]* gate vectors.

The IP packets are sent to the ARP module through the *queueOut* gate. The incoming IP packets are received directly from the network interface cards through the *queueIn[]* gates. Each interface card knows its own network layer gate index.

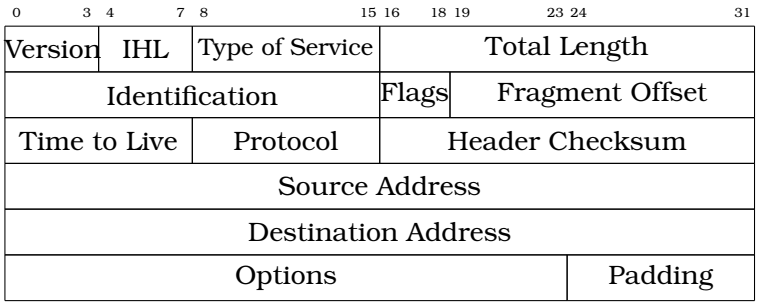
The C++ class of the IPv4 module is derived from QueueBase. There is a processing time associated with each incoming packet. This processing time is specified by the procDelay module parameter. If a packet arrives, when the processing of a previous has not been finished, it is placed in a FIFO queue.

The current performance model assumes that each datagram is processed within the same time, and there is no priority between the datagrams. If you need a more sophisticated performance model, you may change the module implementation (the IP class), and:

1. override the startService() method which determines processing time for a packet, or
2. use a different base class.

11.2.1 IP packets

IP datagrams start with a variable length IP header. The minimum length of the header is 20 bytes, and it can contain at most 40 bytes for options, so the maximum length of the IP header is 60 bytes.



The `Version` field is 4 for IPv4. The 4-bit `IHL` field is the number of 32-bit words in the header. It is needed because the header may contain optional fields, so its length may vary. The minimum IP header length is 20, the maximum length is 60. The header is always padded to multiple of 4 bytes. The `Type of Service` field designed to store priority and preference values of the IP packet, so applications can request low delay, high throughput, and maximium reliability from the routing algorithms. In reality these fields are rarely set by applications, and the routers mostly ignore them. The `Total Length` field is the length of the whole datagram in bytes. The `Identification` field is used for identifying the datagram sent by a host. It is usually generated by incrementing a counter for each outgoing datagram. When the datagram gets fragmented by a router, its `Identification` field is kept unchanged to the other end can collect them. In datagram fragments the `Fragment Offset` is the address of the fragment in the payload of the original datagram. It is measured in 8-byte units, so fragment lengths must be a multiple of 8. Each fragment except the last one, has its `MF` (more fragments) bit set in the `Flags` field. The other used flag in `Flags` is the `DF` (don't fragment) bit which forbids the fragmentation of the datagram. The `Time to Live` field is decremented by each router in the path, and the datagram is dropped if it reached 0. Its purpose is to prevent endless cycles if the routing tables are not properly configured, but can be used for limiting hop count range of the datagram (e.g. for local broadcasts, but the `traceroute` program uses this field too). The `Protocol` field is for demultiplexing the payload of the IP datagram to higher level protocols. Each transport protocol has a registered protocol identifier. The `Header Checksum` field is the 16-bit one's complement sum of the header fields considered as a sequence of 16-bit numbers. The `Source Address` and `Destination Address` are the IPv4 addresses of the source and destination respectively.

The `Options` field contains 0 or more IP options. It is always padded with zeros to a 32-bit boundary. An option is either a single-byte option code or an option code + option length followed by the actual values for the option. Thus IP implementations can skip unknown options.

An IP datagram is represented by the `IPv4Datagram` message class. It contains variables corresponding the fields of the IP header, except:

- `Header Checksum` omitted, modeled by error bit of packets
- `Options` only the following options are permitted and the datagram can contain at most one option:
  - Loose Source Routing
  - Strict Source Routing
  - Timestamp
  - Record Route

The Type of Service field is called diffServCodePoint in IPv4Datagram.

Before sending the IPv4Datagram through the network, the IPv4 module attaches a IPv4RoutingDecision control info. The control info contains the IP address of the next hop, and the identifier of the interface it should be sent. The ARP module translate the IP address to the hardware address on the local net of the specified interface and forwards the datagram to the interface card.

11.2.2 Interface with higher layer

Higher layer protocols should be connected to the transportIn/transportOut gates of the IPv4 module.

Sending packets

Higher layer protocols can send a packet by attaching a IPv4ControlInfo object to their packet and sending it to the IPv4 module.

The following fields must be set in the control info:

- procotol: the Protocol field of the IP datagram. Valid values are defined in the IPProtocolId enumeration.
- destAddr: the Destination Address of the IP datagram.

Optionally the following fields can be set too:

- scrAddr: Source Address of the IP datagram. If given it must match with the address of one of the interfaces of the node, but the datagram is not necessarily routed through that interface. If left unspecified, then the address of the outgoing interface will be used.
- timeToLive: TTL of the IP datagram or -1 (unspecified). If unspecified then the TTL of the datagram will be 1 for destination addresses in the 224.0.0.0 – 224.0.0.255 range. (Datagrams with these special multicast addresses do not need to go further that one hop, routers does not forward these datagrams.) Otherwise the TTL field is determined by the defaultTimeToLive or defaultMCTimeToLive module parameters depending whether the destination address is a multicast address or not.
- dontFragment: the Don't Fragment flag of the outgoing datagram (default is **false**)
- diffServCodePoint: the Type of Service field of the outgoing datagram. (ToS is called diffServCodePoint in IPv4Datagram too.)
- multicastLoop: if **true**, then a copy of the multicast datagrams are sent to the loopback interface, so applications on the same host can receive it.
- interfaceId: id of outgoing interface (can be used to limit broadcast or restrict routing).
- nextHopAddr: explicit routing info, used by Manet DSR routing. If specified, then interfaceId must also be specified. Ignored in Manet routing is disabled.

The IP module encapsulates the transport layer datagram into an IPv4Datagram and fills in the header fields according to the control info. The Identification field is generated by incrementing a counter.

The generated IP datagram is passed to the routing algorithm. The routing decides if the datagram should be delivered locally, or passed to one of the network interfaces with a specified next hop address, or broadcasted on one or all of the network interfaces. The details of the routing is described in the next subsection (11.2.3) in detail.

Before sending the datagram on a specific interface, the `IPv4` module checks if the packet length is smaller than the `MTU` of the interface. If not, then the datagram is fragmented. When the `Don't Fragment` flag forbids fragmentation, an `Destination Unreachable` ICMP error is generated with the `Fragmentation Error (5)` error code.

> **NOTE:** Each fragment will encapsulate the whole higher layer datagram, although the length of the IP datagram corresponds to the fragment length.

The fragments are sent to the `ARP` module through the `queueOut` gate. The `ARP` module forwards the datagram immediately to point-to-point interface cards. If the outgoing interface is a 802.x card, then before forwarding the datagram it performs address resolution to obtain the MAC address of the destination.

Receiving packets

The `IPv4` module of hosts processes the datagrams received from the network in three steps:

1. Reassemble fragments
2. Decapsulate the transport layer datagram
3. Dispatch the datagram to the appropriate transport protocol

When a fragment received, it is added to the fragment buffer of the IP. If the fragment was the last fragment of a datagram, the processing of the datagram continues with step 2. The fragment buffer stores the reception time of each fragment. Fragments older than `fragmentTimeout` are purged from the buffer. The default value of the timeout is 60s. The timeout is only checked when a fragment is received, and at least 10s elapsed since the last check.

An `IPv4ControlInfo` attached to the decapsulated transport layer packet. The control info contains fields copied from the IP header (source and destination address, protocol, TTL, ToS) as well as the interface id through it was received. The control info also stores the original IP datagram, because the transport layer might signal an ICMP error, and the ICMP packet must encapsulate the erronous IP datagram.

> **NOTE:** IP datagrams containing a DSR packet are not decapsulated, the unchanged IP datagram is passed to the DSR module instead.

After decapsulation, the transport layer packet will be passed to the appropriate transport protocol. It must be connected to one of the `transportOut[]` gate. The `IPv4` module finds the gate using the `protocol id→gate index` mapping given in the `protocolMapping` string parameter. The value must be a comma separated list of "<protocol_id>:<gate_index>" items. For example the following line in the ini file maps TCP (6) to gate 0, UDP (17) to gate 1, ICMP (1) to gate 2, IGMP (2) to gate 3, and RVSP (46) to gate 4.

```
**.ip.protocolMapping="6:0,17:1,1:2,2:3,46:4"
```

If the protocol of the received IP datagram is not mapped, or the gate is not connected, the datagram will be silently dropped.

Some protocols are handled differently:

- ICMP: ICMP errors are delivered to the protocol whose packet triggered the error. Only ICMP query requests and responses are sent to the ICMP module.
- IP: sent through preRoutingOut gate. (bug!)
- DSR: ??? (subsection about Manet routing?)

11.2.3 Routing, and interfacing with lower layers

The output of the network interfaces are connected to the queueIn gates of the IPv4 module. The incoming packets are either IP datagrams or ARP responses. The IP datagrams are processed by the IPv4 module, the ARP responses are forwarded to the ARP.

The IPv4 module first checks the error bit of the incoming IP datagrams. There is a *headerlength/packetlength* probability that the IP header contains the error (assuming 1 bit error). With this probability an ICMP Parameter Problem generated, and the datagram is dropped.

When the datagram does not contain error in the IP header, a routing decision is made. As a result of the routing the datagram is either delivered locally, or sent out one or more output interface. When it is sent out, the routing algorithm must compute the next hop of its route. The details are differ, depending on that the destination address is multicast address or not.

When the datagram is decided to be sent up, it is processed as described in the previous subsection (Receiving packets). If it is decided to be sent out through some interface, it is actually sent to the ARP module through the queueOut gate. An IPv4RoutingDecision control info is attached to the outgoing packet, containing the outgoing interface id, and the IP address of the next hop. The ARP module resolve the IP address to a hardware address if needed, and forwards the datagram to next hop.

Unicast/broadcast routing

When the higher layer generated the datagram, it will be processed in these steps:

1. If the destination is the address of a local interface, then the datagram is locally delivered.
2. If the destination is the limited broadcast address, or a local broadcast address, then it will be broadcasted on one or more interface. If the higher layer specified am outgoing interface (interfaceId in the control info), then it will be broadcasted on that interface only. Otherwise if the forceBroadcast module parameter is **true**, then it will broadcasted on all interfaces including the loopback interface. The default value of the forceBroadcast is **false**.
3. If the higher layer provided the routing decision (Manet routing), then the datagram will be sent through the specified interface to the specified next hop.
4. Otherwise IP finds the outgoing interface and the address of the next hop by consulting the routing table, and sends the datagram to the next hop. If no route found, then a Destination Unreachable ICMP error is generated.

Incoming datagrams having unicast or broadcast destination addresses are routed in the following steps:

1. Deliver datagram locally. If the destination address is a local address, the limited broadcast address (255.255.255.255), or a local broadcast address, then it will be sent to the transport layer.

2. Drop packets received from the network when IP forwarding is disabled.

3. Forward the datagram to the next hop. The next hop is determined by looking up the best route to the destination from the routing table. If the gateway is set in the route, then the datagram will be forwarded to the gateway, otherwise it is sent directly to the destination. If no route is found, then a `Destination Unreachable` ICMP error is sent to the source of the datagram.

Multicast routing

Outgoing multicast datagrams are handled as follows:

1. If the higher layer set the `multicastLoop` variable to **`true`**, the IP will send up a copy of the datagram through the loopback interface.

2. Determine the outgoing interface for the multicast datagram, and send out the datagram through that interface. The outgoing interface is determined by the following rules:
   (a) if the HL specified the outgoing interface in the control info, the it will be used
   (b) otherwise use the interface of the route configured in the routing table for the destination address
   (c) if no route found, then use the interface whose address matches the source address of the datagram
   (d) if the HL did not specify the source address, then use the first multicast capable interface
   (e) if no such interface found, then the datagram is unroutable and droppped

Incoming multicast datagrams are forwarded according to their source address (Reverse Path Forwarding), i.e. datagrams are sent away from their sources instead towards their destinations. The multicast routing table maintains a spanning tree for each source network and multicast group. The source network is the root of the tree, and there is a path to each LAN that has members of the multicast group. Each node expects the multicast datagram to arrive from their parent and forwards them towards their children. Multicast forwarding loops are avoided by dropping the datagrams not arrived on the parent interface.

More specifically, the routing routine for multicast datagrams performs these steps:

1. Deliver a copy of the datagram locally. If the interface on which the datagram arrived belongs to the multicast group specified by the destination address, it is sent up to the transport layer.

2. Discard incoming packets that can not be delivered locally and can not be forwarded. A non-local packet can not be forwarded if multicast forwarding is disabled, the destination is a link local multicast address (224.0.0.x), or the TTL field reached 0.

3. Discard the packet if no multicast route found, or if it did not arrive on the parent interface of the route (to avoid multicast loops). If the parent is not set in the route, then the shortest path interface to the source is assumed.

4. Forward the multicast datagram. A copy of the datagram is sent on each child interface described by multicast routes (except the incoming interface). Interfaces may have a

ttlThreshold parameter, that limits the scope of the multicast: only datagrams with higher TTL are forwarded.

11.2.4 Parameters

The IPv4 module has the following parameters:

- procDelay processing time of each incoming datagram.
- timeToLive default TTL of unicast datagrams.
- multicastTimeToLive default TTL of multicast datagrams.
- protocolMapping string value containing the protocol id → gate index mapping, e.g. "6:0,17:1,1:2,2:3,46:4".
- fragmentTimeout the maximum duration until fragments are kept in the fragment buffer.
- forceBroadcast if **true**, then link-local broadcast datagrams are sent out through each interface, if the higher layer did not specify the outgoing interface.

11.2.5 Statistics

The IPv4 module does not write any statistics into files, but it has some statistical information that can be watched during the simulation in the gui environment.

- numForwarded: number of forwarded datagrams, i.e. sent to one of the interfaces (not broadcast), counted before fragmentation.
- numLocalDeliver: number of datagrams locally delivered. (Each fragment counted separately.)
- numMulticast: number of routed multicast datagrams.
- numDropped number of dropped packets. Either because there is no any interface, the interface is not specified and no forceBroadcast, or received from the network but IP forwarding disabled.
- numUnroutable: number of unroutable datagrams, i.e. there is no route to the destination. (But if outgoing interface is specified it is routed!)

In the graphical interface the bubble of the IPv4 module also displays these counters.

11.3 The RoutingTable module

The RoutingTable module represents the routing table. IP hosts and routers contain one instance of this class. It has methods to manage the routing table and the interface table, so one can achieve functionality similar to the route and ifconfig commands.

This is a simple module without gates, it requires function calls to it (message handling does nothing). Methods are provided for reading and updating the interface table and the route table, as well as for unicast and multicast routing.

Interfaces are dynamically registered: at the start of the simulation, every L2 module adds its own interface entry to the table.

The route table is read from a file; the file can also fill in or overwrite interface settings. The route table can also be read and modified during simulation, typically by routing protocol implementations (e.g. OSPF).

Entries in the route table are represented by `IPv4Route` objects. `IPv4Route` objects can be polymorphic: if a routing protocol needs to store additional data, it can simply subclass from `IPv4Route`, and add the derived object to the table. The `IPv4Route` object has the following fields:

- `host` is the IP address of the target of the route (can be a host or network). When an entry searched for a given destination address, the destination address is compared with this `host` address using the `netmask` below, and the longest match wins.
- `netmask` used when comparing `host` with the detination address. It is 0.0.0.0 for the default route, 255.255.255.255 for host routes (exact match), or the network or subnet mask for network routes.
- `gateway` is the IP address of the gateway for indirect routes, or 0.0.0.0 for direct routes. Note that 0.0.0.0 can be used even if the destination is not directly connected to this node, but can be found using proxy ARP.
- `interface` the outgoing interface to be used with this route.
- `type` `DIRECT` or `REMOTE`. For direct routes, the next hop address is the destination address, for remote routes it is the gateway address.
- `source` `MANUAL`, `IFACENETMASK`, `RIP`, `OSPF`, `BGP`, `ZEBRA`, `MANET`, or `MANET2`. `MANUAL` means that the route was added by a routing file, or a network configurator. `IFACENETMASK` routes are added for each interface of the node. Other values means that the route is managed by the specific routing daemon.
- `metric` the "cost" of the route. Currently not used when choosing the best route.

In multicast routers the routing table contains multicast routes too. A multicast route is represented by an instance of the `IPv4MulticastRoute` class. The `IPv4MulticastRoute` instance stores the following fields:

- `origin` IP address of the network of the source of the datagram
- `originNetmask` netmask of the source network
- `group` the multicast group to be matched the destination of the datagram. If unspecified, then the route matches with
- `parent` interface towards the parent link in the multicast tree. Only those datagrams are forwarded that arrived on the parent interface.
- `children` the interfaces on which the multicast datagram to be forwarded. Each entry contains a flag indicating if this interface is a leaf in the multicast tree. The datagram is forwarded to leaf interfaces only if there are known members of the group in the attached LAN.
- `source` enumerated value identifying the creator of the entry. `MANUAL` for static routes, `DVRMP` for the DVMRP routers, `PIM_SM` for PIM SM routers.

- `metric` the “cost“ of the route.

When there are several multicast routes matching the source and destination of the datagram, then the forwarding algorithm chooses the one with the

1. the longest matching source
2. the more specific group
3. the smallest metric.

Parameters

The `RoutingTable` 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
- `IPForward`: turns IP forwarding on/off (It is always **`true`** in a `Router` and is **`false`** by default in a `StandardHost`.)
- `forwardMulticast`: turns multicast IP forwarding on/off. Default is **`false`**, should be set to **`true`** in multicast routers.
- `routingFile`: name of routing file that configures IP addresses and routes of the node containing this routing table. Its format is described in section 11.9.3.

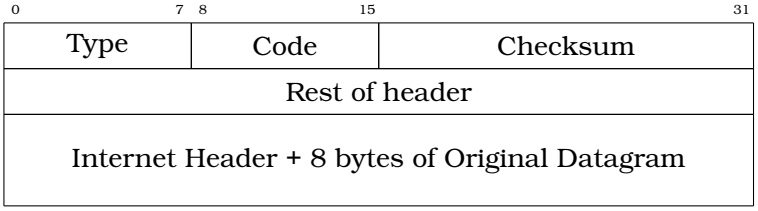
WARNING: The `routingFile` parameter is obsolete. The prefered method for network configuration is to use `IPv4NetworkConfigurator`. The old config files should be replaced with the XML configuration of `IPv4NetworkConfigurator`. Section 11.9.1 describes the format of the new configuration files.

11.4 The ICMP module

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

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

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



The corresponding message class (ICMPMessage) contains only the Type and Code fields. The message encapsulates the IP packet that triggered the error, or the data of the ping request/reply.

The ICMP module has two methods which can be used by other modules to send ICMP error messages:

- sendErrorMessage(IPv4Datagram*, ICMPType, ICMPCode) used by the network layer to report erronous IPv4 datagrams. The ICMP header fields are set to the given type and code, and the ICMP message will encapsulate the given datagram.
- sendErrorMessage(cPacket*, IPv4ControlInfo*, ICMPType, ICMPCode) used by the transport layer components to report erronous packets. The transport packet will be encapsulated into an IP datagram before wrapping it into the ICMP message.

The ICMP module can be accessed from other modules of the node by calling ICMPAccess::get().

When an incoming ICMP error message is received, the ICMP module sends it out on the errorOut gate unchanged. It is assumed that an external module is connected to errOut that can process the error packet. There is a simple module (ErrorHandling) that simply logs the error and drops the message. Note that the IPv4 module does not send REDIRECT, DESTINATION_UNREACHABLE, TIME_EXCEEDED and PARAMETER_PROBLEM messages to the ICMP module, it will send them to the transport layer module that sent the bogus packet encapsulated in the ICMP message.

> **NOTE:** ICMP protocol encapsulates only the IP header + 8 byte following the IP header from the bogus IP packet. The ICMP packet length computed from this truncated packet, despite it encapsulates the whole IP message object. As a consequence, calling decapsulate() on the ICMP message will cause an "packet length became negative" error. To avoid this, use getEncapsulatedMsg() to access the IP packet that caused the ICMP error.

The ICMP module receives ping commands on the pingIn gate from the application. The ping command can be any packet having an IPv4ControlInfo control info. The packet will be encapsulated with an ICMPMessage and handed over to the IP.

If ICMP receives an echo request from IP, the original message object will be returned as the echo reply. Of course, before sending back the object to IP, the source and destination addresses are swapped and the message type changed to ICMP_ECHO_REPLY.

When an ICMP echo reply received, the application message decapsulated from it and passed to the application through the pingOut gate. The IPv4ControlInfo also copied from the ICMPMessage to the application message.

11.5 The ARP module

The ARP module implements the Address Resolution Protocol (RFC826). The ARP protocol is designed to translate a local protocol address to a hardware address. Altough the ARP protocol can be used with several network protocol and hardware addressing schemes, in practice they are almost always IPv4 and 802.3 addresses. The INET implementation of the ARP protocol (the ARP module) supports only IP address → MAC address translation.

If a node wants to send an IP packet to a node whose MAC address is unknown, it broadcasts an ARP frame on the Ethernet network. In the request its publish 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.

The frame format of the ARP request and reponse is shown in Figure 11.5. In our case the HTYPE (hardware type), PTYPE (protocol type), HLEN (hardware address length) and PLEN (protocol address length) are constants: HTYPE=Ethernet (1), PTYPE=IPv4 (2048), HLEN=6, PLEN=4. The OPER (operation) field is 1 for an ARP request and 2 for an ARP response. The SHA field contains the 48-bit hardware address of the sender, SPA field is the 32-bit IP address of the sender; THA and TPA are the addresses of the target. The message class corresponding to the ARP frame is `ARPPacket`. In this class only the OPER, SHA, SPA, THA and TPA fields are stored. The length of an `ARPPacket` is 28 bytes.

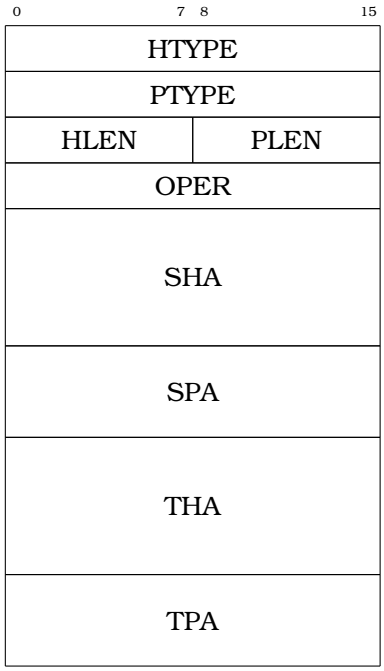


Figure 11.1: ARP frame

The `ARP` module receives IP datagrams and ARP responses from `IPv4` on the `ipIn` gate and transmits IP datagrams and ARP requests on the `nicOut[]` gates towards the network interface cards. ARP broadcasts the requests on the local network, so the NIC's entry in the `InterfaceTable` should have `isBroadcast()` flag set in order to participate in the address resolution.

The incoming IP packet should have an attached `IPv4RoutingDecision` control info containing the IP address of the next hop. The next hop 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 mapped using the ARP procotol.

If the hardware 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. The `ARP` module creates an `ARPPacket` object, sets the sender MAC and IP address to its own address, sets the destination IP address to the address of the target of

the IP datagram, leave the destination MAC address blank and broadcasts the packet on each network interface with broadcast capability. Before sending the ARP packet, it retransmission a timer. If the timer expires, it will retransmit the ARP request, until the maximum retry count is reached. If there is no response to the ARP request, then the address resolution fails, and the IP packet is dropped from the queue. Otherwise the MAC address of the destination is learned and the IP packet can be transmitted on the corresponding interface.

When an ARP packet is received on the `ipIn` gate, and the sender's IP is already in the ARP cache, it is updated with the information in the ARP frame. Then it is checked that the destination IP of the packet matches with our address. In this case a new entry is created with the sender addresses in the ARP cache, and if the packet is a request a response is created and sent directly to the originator. If proxy ARP is enabled, the request can be responded with our MAC address if we can route IP packets to the destination.

Usually each `ARP` module maintains a local ARP cache. However it is possible to use a global cache. The global cache is filled in with entries of the IP and MAC addresses of the known interfaces when the ARP modules are initiated (at simulation time 0). `ARP` modules that are using the global ARP cache never initiate an address resolution; if an IP address not found in the global cache, the simulation stops with an error. However they will respond to ARP request, so the simulation can be configured so that some `ARP`s use local, while others the global cache.

When an entry is inserted or updated in the local ARP cache, the simulation time saved in the entry. The mapping in the entry is not used after the configured `cacheTimeout` elapsed. This parameter does not affect the entries of the global cache however.

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`**)

The `ARP` module emits four signals:

- `sentReq`: emits 1 each time an ARP request is sent
- `sentReplies`: emits 1 each time an ARP response is sent
- `initiatedResolution`: emits 1 each time an ARP resolution is initiated
- `failedResolution`: emits 1 each time an ARP resolution is failed

These signals are recorded as vectors and their counts as scalars.

11.6 The IGMP module

The IGMP module is responsible for distributing the information of multicast group memberships from hosts to routers. When an interface of a host joins to 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 has 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 behaviour in host and routers in details. Note that multicast routers behaves as hosts too, i.e. they are sending reports to other routers when joining or leaving a multicast group.

11.6.1 Host behaviour

When an interface joins to a multicast group, the host will send a Membership Report immediately to the group address. This report is repeated after `unsolicetedReportInterval` 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 observe 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.

11.6.2 Router behaviour

Multicast routers maintains 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` 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 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 more 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 in separated by `lastMemberQueryInterval` until a Membership Report is received. If no Report received, then the router assumes that the group has no local members.

11.6.3 Disabling IGMP

The IPv4 `NetworkLayer` contains an instance of the IGMP module. IGMP can be turned off by setting the `enabled` parameter to false. When disabled, then no IGMP message is generated, and incoming IGMP messages are ignored.

11.6.4 Parameters

The following parameters has effects in both hosts and routers:

- `enabled` if **`false`** then the IGMP module is silent. Default is **`true`**.

These parameters are only used in hosts:

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

Router timeouts are configured by these parameters:

- `robustnessVariable` the IGMP is robust to `robustnessVariable`-1 packet losses. Default is 2.
- `queryInterval` the interval between General Queries sent by a Querier. 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. Default is `queryInterval` / 4.
- `startupQueryCount` the number of Queries sent out on startup, separated by the `startupQueryInterval`. 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. Default is 1s.
- `lastMemberQueryCount` the number of Group-Specific Queries sent before the router assumes there are no local members. Default is `robustnessVariable`.

11.7 The NetworkLayer module

The NetworkLayer module packs the IP, ICMP, ARP, and IGMP modules into one compound module. The compound module defines gates for connecting UDP, TCP, SCTP, RSVP and OSPF transport protocols. The pingIn and pingOut gates of the ICMP module are also available, while its errorOut gate is connected to an inner ErrorHandling component that writes the ICMP errors to the log.

The component can be used in hosts and routers to support IPv4.

11.8 The NetworkInfo module

The NetworkInfo module can be used to dump detailed information about the network layer. This module does not send or received messages, it is invoked by the ScenarioManager instead. For example the following ScenarioManager script dump the routing table of the LSR2 module at simulation time $t = 2$ into LSR2_002.txt:

```
<scenario>
  <at t="2">
    <routing module="NetworkInfo" target="LSR2" file="LSR2_002.txt"/>
  </at>
</scenario>
```

The module currently support only the routing command which dumps the routing table. The command has four parameters given as XML attributes:

- target the name of the node that owns the routing table to be dumped
- filename the name of the file the output is directed to
- mode if set to "a", the output is appended to the file, otherwise the target is truncated if the file existed
- compat if set to "linux", then the output is generated in the format of the route -n command of Linux. The output is sorted only if compat is **true**.

11.9 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 a 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 FlatNetworkConfigurator for automatic and routing files for manual configuration. Both 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`, `FlatNetworkConfigurator` and old routing files are described in the following sections.

11.9.1 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) has already registered their interface 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 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 initialize 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 extremums. 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 signalled.

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. The resulting routing tables will have one entry for 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 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 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 a `RoutingTable`. 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 interface connected by a point-to-point connection forms 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>
```

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 allows 2 nodes in the subnet.)

The following configuration generates network addresses below the 10.0.0.0 address for each link, and assign 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 its 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` 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 "*" that matches all hosts. e.g. "subnet.client*" or "host* router[0..3]" or "area*.*.host[0]"
- `@names` Optional selector attribute that specifies a list of interface name patterns. Only interfaces with the specified names are affected. The default value is "*" that matches all interfaces. e.g. "eth* ppp0" or "*"
- `@towards` 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 with 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 "*". e.g. "ap" or "server" or "client*"
- `@among` 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 same as 'hosts="X Y Z" towards="X Y Z"'.
- `@address` Optional parameter attribute that limits the range of assignable addresses. Wildcards are allowed with using 'x' as part of the address in place of a byte. Unspecified

parts will be filled automatically by the configurator. The default value "" 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. e.g. "192.168.1.1" or "10.0.x.x"
- `@netmask` Optional parameter attribute that limits the range of assignable netmasks. Wildcards are allowed with using 'x' as part of the netmask in place of a byte. Unspecified parts will be filled automatically be the configurator. The default value "" means that any netmask can be configured. e.g. "255.255.255.0" or "255.255.x.x" or "255.255.x.0"
- `@mtu` number Optional parameter attribute to set the MTU parameter in the interface. When unspecified the interface parameter is left unchanged.
- `@metric` 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 wireless network, unique value used if missed
- `@hosts` Optional selector attribute that specifies a list of host name patterns. Only interfaces in the specified hosts are affected. The default value is "*" that matches all hosts.
- `@interfaces` Optional selector attribute that specifies a list of interface name patterns. Only interfaces with the specified names are affected. The default value is "*" that matches all interfaces.

Multicast groups

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

For example

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

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` Optional selector attribute that specifies a list of host name patterns. Only interfaces in the specified hosts are affected. The default value is "*" that matches all hosts.
- `@interfaces` Optional selector attribute that specifies a list of interface name patterns. Only interfaces with the specified names are affected. The default value is "*" that matches all interfaces.
- `@towards` 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` 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 same as 'hosts="X Y Z" towards="X Y Z"'.
- `@address` 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. e.g. "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` Optional selector attribute that specifies a list of host name patterns. Only routing tables in the specified hosts are affected. The default value "" means all hosts will be affected. e.g. "host* router[0..3]"
- `@destination` Optional parameter attribute that specifies the destination address in the route (IPvXAddressResolver syntax). The default value is "*". e.g. "192.168.1.1" or "subnet.client[3]" or "subnet.server(ipv4)" or "*"
- `@netmask` Optional parameter attribute that specifies the netmask in the route. The default value is "*". e.g. "255.255.255.0" or "/29" or "*"
- `@gateway` Optional parameter attribute that specifies the gateway (next-hop) address in the route (IPvXAddressResolver syntax). When unspecified the interface parameter must be specified. The default value is "*". e.g. "192.168.1.254" or "subnet.router" or "*"
- `@interface` 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. e.g. "eth0"
- `@metric` 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` Optional selector attribute that specifies a list of host name patterns. Only routing tables in the specified hosts are affected. e.g. "host* router[0..3]"
- `@source` Optional parameter attribute that specifies the address of the source network. The default value is "*" that matches all sources.
- `@netmask` Optional parameter attribute that specifies the netmask of the source network. The default value is "*" that matches all sources.
- `@groups` Optional List of IPv4 multicast addresses specifying the groups this entry applies to. The default value is "*" that matches all multicast groups. e.g. "225.0.0.1 225.0.1.2".
- `@metric` Optional parameter attribute that specifies the metric in the route.
- `@parent` 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` 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 neighbour 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 intsructs the routing algorithm to forward multicast datagrams whose source is in the 10.0.1.0 network and whose destinatation 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 add 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. The resulting routing tables will have one entry for 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.

> **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 `IPv4NetorkConfigurator`:

- `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)

11.9.2 FlatNetworkConfigurator

The `FlatNetworkConfigurator` module configures IP addresses and routes of IP nodes of a network. All assigned addresses share a common subnet prefix, 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 configurator, so it relies on proxy ARP if the network spans several LANs.

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

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

- `netmask` 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 maximum 65534 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.

11.9.3 Old routing files

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

Interfaces themselves are represented in the simulation by modules (such as the PPP module). Modules automatically register themselves with appropriate defaults in the RoutingTable, 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 was 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, that can not be populated from the old routing table entries. Therefore simulations that use multicast forwarding can not 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 was given for the multicast group, then IP sent a copy of the datagram on each interface of that routes. From version 1.99.4, only the first matching interface is used (considering longest match). If the application wants to send the multicast datagram on each interface, then it must explicitly loop and specify the multicast interface.

11.10 Applications

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

11.10.1 IP traffic generators

Traffic generators that connect directly to IP (without using TCP or UDP): `IIPvXTraffixGenerator` (prototype). `IPvXTrafGen`,

Sends IP or IPv6 datagrams to the given address at the given `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 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 `IPvXAddressResolver` 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 `sentPk` 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 `rcvdPacket` 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 `IPvXTrafSink`.

You can see an example usage of these applications in `examples/inet/routerperf/omnetpp.ini` simulaton.

11.10.2 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 `IPvXAddressResolver` class is used to resolve the address.) To disable send, specify empty destAddr.

Every ping request is sent out with a sequence number, and replies are expected to arrive in the same order. Whenever there's 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) is 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 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.)

Uses `PingPayload` as payload for the ICMP(v6) Echo Request/Reply packets.

Parameters

- `destAddr`: destination address
- `srcAddr`: source address (useful with multi-homing)
- `packetSize`: of ping payload, in bytes (default is 56)
- `sendInterval`: time to wait between pings (can be random, default is 1s)
- `hopLimit`: TTL or hopLimit for IP packets (default is 32)
- `count`: stop after `count` ping request, 0 means continuously

- `startTime`: send first ping request at `startTime`
- `stopTime`: time of finish sending, 0 means forever
- `printPing`: dump on stdout (default is **`false`**)

Signals and Statistics

- `rtt` value of the round trip time
- `numLost` number of lost packets
- `outOfOrderArrivals` number of packets arrived out-of-order
- `pingTxSeq` sequence number of the sent ping request
- `pingRxSeq` sequence number of the received ping response

Chapter 12

IPv6 and Mobile IPv6

12.1 Overview

IPv6 support is implemented by several cooperating modules. The IPv6 module implements IPv6 datagram handling (sending, forwarding etc). It relies on `RoutingTable6` to get access to the routes. `RoutingTable6` also contains the neighbour discovery data structures (destination cache, neighbour cache, prefix list – the latter effectively merged into the route table). Interface configuration (address, state, timeouts etc) is held in the `InterfaceTable`, in `IPv6InterfaceData` objects attached to `InterfaceEntry` as its `ipv6()` member.

The module `IPv6NeighbourDiscovery` implements all tasks associated with neighbour discovery and stateless address autoconfiguration. The data structures themselves (destination cache, neighbour cache, prefix list) are kept in `RoutingTable6`, and are accessed via public C++ methods. Neighbour discovery packets are only sent and processed by this module – when IPv6 receives one, it forwards the packet to `IPv6NeighbourDiscovery`.

The rest of ICMPv6 (ICMP errors, echo request/reply etc) is implemented in the module `ICMPv6`, just like with IPv4. ICMP errors are sent into `IPv6ErrorHandling`, which the user can extend or replace to get errors handled in any way they like.

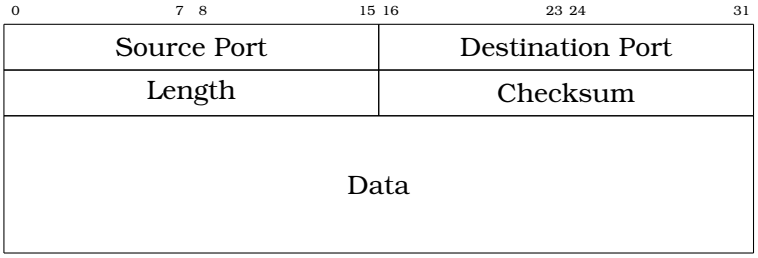
Chapter 13

The UDP Model

13.1 Overview

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

The frame format as described in RFC768:



The ports represents the communication end points that are allocated by the applications that want to send or receive the datagrams. The "Data" field is the encapsulated application data, the "Length" and "Checksum" fields are computed from the data.

The INET framework contains an `UDP` module that performs the encapsulation/decapsulation of user packets, an `UDPSocket` class that provides the application the usual socket interface, and several sample applications.

These components implement the following statndards:

- RFC768: User Datagram Protocol
- RFC1122: Requirements for Internet Hosts – Communication Layers

13.2 The UDP module

The UDP protocol is implemented by the `UDP` simple module. 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.

Each UDP module has gates to connect to the IPv4 and IPv6 network layer (ipIn/ipOut and ipv6In/ipv6Out), and a gate array to connect to the applications (appIn/appOut).

The UDP module can be connected to several applications, and each application can use several sockets to send and receive UDP datagrams. The state of the sockets are stored within the UDP module and the application can configure the socket by sending command messages to the UDP module. These command messages are distinguished by their kind and the type of their control info. The control info identifies the socket and holds the parameters of the command.

Applications don't have to send messages directly to the UDP module, as they can use the `UDPSocket` utility class, which encapsulates the messaging and provides a socket like interface to applications.

13.2.1 Sending UDP datagrams

If the application want to send datagrams, it optionally can connect to the destination. It does this be sending a message with UDP_C_CONNECT kind and `UDPConnectCommand` control info containing the remote address and port of the connection. The UDP protocol is in fact connectionless, so it does not send any packets as a result of the connect call. When the UDP module receives the connect request, it simply remembers the destination address and port and use it as default destination for later sends. The application can send several connect commands to the same socket.

For sending an UDP packet, the application should attach an `UDPSendCommand` control info to the packet, and send it to `UDP`. The control info may contain the destination address and port. If the destination address or port is unspecified in the control info then the packet is sent to the connected target.

The `UDP` module encapsulates the application's packet into an `UDPPacket`, creates an appropriate IP control info and send it over ipOut or ipv6Out depending on the destination address.

The destination address can be the IPv4 local broadcast address (255.255.255.255) or a multicast address. Before sending broadcast messages, the socket must be configured for broadcasting. This is done by sending an message to the UDP module. The message kind is UDP_C_SETOPTION and its control info (an `UDPSetBroadcastCommand`) tells if the broadcast is enabled. You can limit the multicast to the local network by setting the TTL of the IP packets to 1. The TTL can be configured per socket, by sending a message to the UDP with an `UDPSetTimeToLive` control info containing the value. If the node has multiple interfaces, the application can choose which is used for multicast messages. This is also a socket option, the id of the interface (as registered in the interface table) can be given in an `UDPSetMulticastInterfaceCommand` control info.

> **NOTE:** The `UDP` module supports only local broadcasts (using the special 255.255.255.255 address). Packages that are broadcasted to a remote subnet are handled as undeliverable messages.

If the UDP packet cannot be delivered because nobody listens on the destination port, the application will receive a notification about the failure. The notification is a message with UDP_I_ERROR kind having attached an `UDPErrorIndication` control info. The control info contains the local and destination address/port, but not the original packet.

After the application finished using a socket, it should close it by sending a message UDP_C_CLOSE kind and `UDPCloseCommand` control info. The control info contains only the

socket identifier. This command frees the resources associated with the given socket, for example its socket identifier or bound address/port.

13.2.2 Receiving UDP datagrams

Before receiving UDP datagrams applications should first "bind" to the given UDP port. This can be done by sending a message with message kind UDP_C_BIND attached with an `UDPBindCommand` control info. The control info contains the socket identifier and the local address and port the application want to receive UDP packets. Both the address and port is optional. If the address is unspecified, than the UDP packets with any destination address is passed to the application. If the port is -1, then an unused port is selected automatically by the UDP module. The localAddress/localPort combination must be unique.

When a packet arrives from the network, first its error bit is checked. Erronous messages are dropped by the UDP component. Otherwise the application bound to the destination port is looked up, and the decapsulated packet passed to it. If no application is bound to the destination port, an ICMP error is sent to the source of the packet. If the socket is connected, then only those packets are delivered to the application, that received from the connected remote address and port.

The control info of the decapsulated packet is an `UDPDataIndication` and contains information about the source and destination address/port, the TTL, and the identifier of the interface card on which the packet was received.

The applications are bound to the unspecified local address, then they receive any packets targeted to their port. UDP also supports multicast and broadcast addresses; if they are used as destination address, all nodes in the multicast group or subnet receives the packet. The socket receives the broadcast packets only if it is configured for broadcast. To receive multicast messages, the socket must join to the group of the multicast address. This is done be sending the UDP module an UDP_C_SETOPTION message with `UDPJoinMulticastGroupsCommand` control info. The control info specifies the multicast addresses and the interface identifiers. If the interface identifier is given only those multicast packets are received that arrived at that interface. The socket can stop receiving multicast messages if it leaves the multicast group. For this purpose the application should send the UDP another UDP_C_SETOPTION message in their control info (`UDPLeaveMulticastGroupsCommand`) specifying the multicast addresses of the groups.

13.2.3 Signals

The `UDP` module emits the following signals:

- `sentPk` when an UDP packet sent to the IP, the packet
- `rcvdPk` when an UDP packet received from the IP, the packet
- `passedUpPk` when a packet passed up to the application, the packet
- `droppedPkWrongPort` when an undeliverable UDP packet received, the packet
- `droppedPkBadChecksum` when an erronous UDP packet received, the packet

13.3 UDP sockets

UDPSocket is a convenience class, to make it easier to send and receive UDP packets from your application models. You'd have one (or more) UDPSocket object(s) in your application simple module class, and call its member functions (bind(), connect(), sendTo(), etc.) to create and configure a socket, and to send datagrams.

UDPSocket chooses and remembers the sockId for you, assembles and sends command packets such as UDP_C_BIND to UDP, and can also help you deal with packets and notification messages arriving from UDP.

Here is a code fragment that creates an UDP socket and sends a 1K packet over it (the code can be placed in your handleMessage() or activity()):

```
UDPSocket socket;
socket.setOutputGate(gate("udpOut"));
socket.connect(IPvXAddress("10.0.0.2"), 2000);

cPacket *pk = new cPacket("dgram");
pk->setByteLength(1024);
socket.send(pk);

socket.close();
```

Processing messages sent up by the UDP module is relatively straightforward. You only need to distinguish between data packets and error notifications, by checking the message kind (should be either UDP_I_DATA or UDP_I_ERROR), and casting the control info to UDPDataIndication or UDPErrorIndication. USPSocket provides some help for this with the `belongsToSocket()` and `belongsToAnyUDPSocket()` methods.

```
void MyApp::handleMessage(cMessage *msg)
{
    if (msg->getKind() == UDP_I_DATA)
    {
        if (socket.belongsToSocket())
            processUDPPacket(PK(msg));
    }
    else if (msg->getKind() == UDP_I_ERROR)
    {
        processUDPError(msg);
    }
    else
    {
        error("Unrecognized message (%s)", msg->getClassName());
    }
}
```

13.4 UDP applications

All UDP applications should be derived from the `IUDPApp` module interface, so that the application of `StandardHost` could be configured without changing its NED file.

The following 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 similar to UDPBasicApp, but it sends back the packet after reception
- UDPSink consumes and prints packets received from the UDP module
- UDPVideoStreamCli,UDPVideoStreamSvr simulates UDP streaming

The next sections describe these applications in details.

13.4.1 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 message and the message length can be given as a random variable. Before the packet is sent, it is emitted in the sentPk signal.

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

The number of sent and received messages are saved as scalars at the end of the simulation.

13.4.2 UDPSink

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

13.4.3 UDPEchoApp

Similar to UDPBasicApp, but it sends back the packet after reception. It accepts only packets with UDPEchoAppMsg type, i.e. packets that are generated by another UDPEchoApp.

When an echo response received, it emits an roundTripTime signal.

13.4.4 UDPVideoStreamCli

This module is a video streaming client. It send one “video streaming request” to the server at time startTime and receives stream from UDPVideoStreamSvr.

The received packets are emitted by the rcvdPk signal.

13.4.5 UDPVideoStreamSvr

This is the video stream server to be used with UDPVideoStreamCli.

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 accomodate jitter.

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

13.4.6 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 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 `IPvXAddressResolver` class is used to resolve the address.) You can use the "Broadcast" string as address for sending broadcast messages.

INET also defines several NED functions that can be useful:

- moduleListByPath("pattern",...):
  Returns a space-separated list of the modulenames. All modules whole getFullPath() matches one of the pattern parameters will get included. The patterns may contain wilcards in the same syntax as in ini files. See cTopology::extractByModulePath() function example: destaddresses = moduleListByPath("**.host[*]", "**.fixhost[*]")

- moduleListByNedType("fully.qualified.ned.type",...):
  Returns a space-separated list of the modulenames with the given NED type(s). All modules whose getNedTypeName() is listed in the given parameters will get included. The NED type name is fully qualified. See cTopology::extractByNedTypeName() function example: 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 burst is the `sleepDuration` parameter; this can be zero (zero is not allowed for `sendInterval`.) The zero `burstDuration` is interpreted as infinity.

Packets

Packet length is controlled by the `messageLength` parameter.

The module adds two parameters to packets before sending:

- sourceID: source module ID
- msgId: incremented by 1 after send any packet.

When received packet has this parameters, the module checks the order of received packets.

Operation as sink

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

Statistics

Statistics are collected on outgoing packets:

- sentPk: packet object

Statistics are collected on incoming packets:

- outOfOrderPk: statistics of out of order packets. The packet is out of order, when has msgId and sourceId parameters and module received bigger msgId from same sourceID.
- dropPk: statistics of dropped packets. The packet is dropped when not out-of-order packet and delay time is larger than delayLimit parameter. The delayLimit=0 is infinity.
- rcvdPk: statistics of not dropped, not out-of-order packets.
- endToEndDelay: end to end delay statistics of not dropped, not out-of-order packets.

Chapter 14

The TCP Models

14.1 Overview

TCP protocol is the most widely used protocol of the Internet. It provides reliable, ordered delivery of stream of bytes from one application on one computer to another application on another computer. It is used by such applications as World Wide Web, email, file transfer amongst others.

The baseline TCP protocol is described in RFC793, but other tens of RFCs contains modifications and extensions to the TCP. These proposals enhance the efficiency and safety of the TCP protocol and they are widely implemented in the real TCP modules. As a result, TCP is a complex protocol and sometimes it is hard to see how the different requirements interacts with each other.

The TCP modules of the INET framework implements the following RFCs:

RFC 793	Transmission Control Protocol
RFC 896	Congestion Control in IP/TCP Internetworks
RFC 1122	Requirements for Internet Hosts – Communication Layers
RFC 1323	TCP Extensions for High Performance
RFC 2018	TCP Selective Acknowledgment Options
RFC 2581	TCP Congestion Control
RFC 2883	An Extension to the Selective Acknowledgement (SACK) Option for TCP
RFC 3042	Enhancing TCP's Loss Recovery Using Limited Transmit
RFC 3390	Increasing TCP's Initial Window
RFC 3517	A Conservative Selective Acknowledgment (SACK)-based Loss RecoveryAlgorithm for TCP
RFC 3782	The NewReno Modification to TCP's Fast Recovery Algorithm

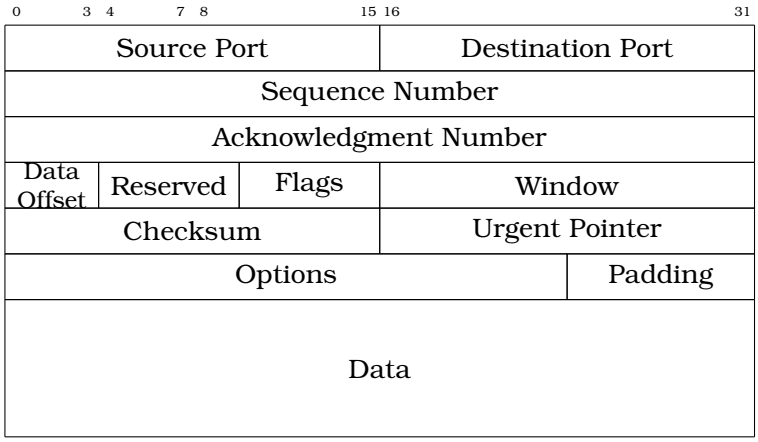
In this section we describe the features of the TCP protocol specified by these RFCs, the following sections deal with the implementation of the TCP in the INET framework.

14.1.1 TCP segments

The TCP module transmits a stream of the data over the unreliable, datagram service that the IP layer provides. When the application writes a chunk of data into the socket, the TCP module breaks it down to packets and hands it over the IP. On the receiver side, it collects the recieved packets, order them, and acknowledges the reception. The packets that are not acknowledged in time are retransmitted by the sender.

The TCP procotol can address each byte of the data stream by *sequence numbers*. The sequence number is a 32-bit unsigned integer, if the end of its range is reached, it is wrapped around.

The layout of the TCP segments is described in RFC793:



Here

- the Source and Destination Ports, together with the Source and Destination addresses of the IP header identifies the communication endpoints.
- the Sequence Number identifier of the first data byte transmitted in the sequence, Sequence Number + 1 identifies the second byte, so on. If the SYN flag is set it consumes one sequence number before the data bytes.
- the Acknowlegment Number refers to the next byte (if the ACK flag is set) expected by the receiver using its sequence number
- the Data Offset is the length of the TCP header in 32-bit words (needed because the Options field has variable length)
- the Reserved bits are unused
- the Flags field composed of 6 bits:
  - URG: Urgent Pointer field is significant
  - ACK: Acknowledgment field is significant
  - PSH: Push Function
  - RST: Reset the connection
  - SYN: Synchronize sequence number
  - FIN: No more data from sender
- the Window is the number of bytes the receiver TCP can accept (because of its limited buffer)
- the Checksum is the 1-complement sum of the 16-bit words of the IP/TCP header and data bytes
- the Urgent Pointer is the offset of the urgent data (if URG flag is set)
- the Options field is variable length, it can occupy 0-40 bytes in the header and is always padded to a multiple of 4 bytes.

14.1.2 TCP connections

When two applications are communicating via TCP, one of the applications is the client, the other is the server. The server usually starts a socket with a well known local port and waits until a request comes from clients. The client applications are issue connection requests to the port and address of the service they want to use.

After the connection is established both the client and the server can send and receive data. When no more data is to be sent, the application closes the socket. The application can still receive data from the other direction. The connection is closed when both communication partner closed its socket.

...

When opening the connection an initial sequence number is choosen and communicated to the other TCP in the SYN segment. This sequence number can not be a constant value (e.g. 0), because then data segments from a previous incarnation of the connection (i.e. a connection with same addresses and ports) could be erronously accepted in this connection. Therefore most TCP implementation choose the initial sequence number according to the system clock.

14.1.3 Flow control

The TCP module of the receiver buffers the data of incoming segments. This buffer has a limited capacity, so it is desirable to notify the sender about how much data the client can accept. The sender stops the transmission if this space exhausted.

In TCP every ACK segment holds a Window field; this is the available space in the receiver buffer. When the sender reads the Window, it can send at most Window unacknowledged bytes.

Window Scale option

The TCP segment contains a 16-bit field for the Window, thus allowing at most 65535 byte windows. If the network bandwidth and latency is large, it is surely too small. The sender should be able to send bandwitdh*latency bytes without receiving ACKs.

For this purpose the Window Scale (WS) option had been introduced in RFC1323. This option specifies a scale factor used to interpret the value of the Window field. The format is the option is:

If the TCP want to enable window sizes greater than 65535, it should send a WS option in the SYN segment or SYN/ACK segment (if received a SYN with WS option). Both sides must send the option in the SYN segment to enable window scaling, but the scale in one direction might differ from the scale in the other direction. The $shift.cnt$ field is the 2-base logarithm of the window scale of the sender. Valid values of $shift.cnt$ are in the $[0, 14]$ range.

Persistence timer

When the reciever buffer is full, it sends a 0 length window in the ACK segment to stop the sender. Later if the application reads the data, it will repeat the last ACK with an updated

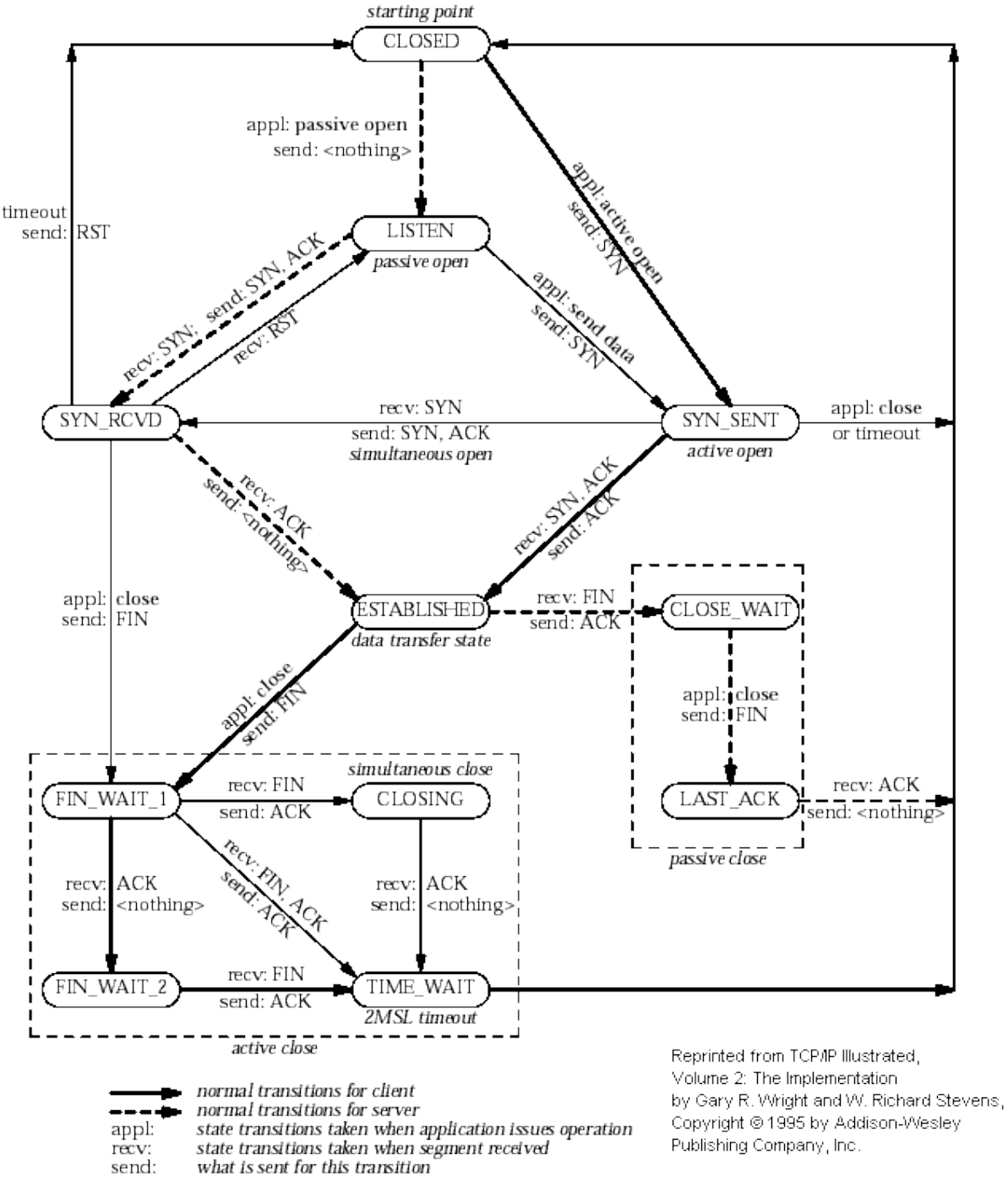


Figure 14.1: TCP state diagram

window to resume data sending. If this ACK segment is lost, then the sender is not notified, so a deadlock happens.

To avoid this situation the sender starts a Persistence Timer when it received a 0 size window. If the timer expires before the window is increased it send a probe segment with 1 byte of data. It will receive the current window of the receiver in the response to this segment.

Keepalive timer

TCP keepalive timer is used to detect dead connections.

14.1.4 Transmission policies

Retransmissions

When the sender TCP sends a TCP segment it starts a retransmission timer. If the ACK arrives before the timer expires it is stopped, otherwise it triggers a retransmission of the segment.

If the retransmission timeout (RTO) is too high, then lost segments causes high delays, if it is too low, then the receiver gets too many useless duplicated segments. For optimal behaviour, the timeout must be dynamically determined.

Jacobson suggested to measure the RTT mean and deviation and apply the timeout:

$$RTO = RTT + 4 * D$$

Here RTT and D are the measured smoothed roundtrip time and its smoothed mean deviation. They are initialized to 0 and updated each time an ACK segment received according to the following formulas:

$$RTT = \alpha * RTT + (1 - \alpha) * M$$

$$D = \alpha * D + (1 - \alpha) * |RTT - M|$$

where M is the time between the segments send and the acknowledgment arrival. Here the α smoothing factor is typically $7/8$.

One problem may occur when computing the round trip: if the retransmission timer timed out and the segment is sent again, then it is unclear that the received ACK is a response to the first transmission or to the second one. To avoid confusing the RTT calculation, the segments that have been retransmitted do not update the RTT. This is known as Karn's modification. He also suggested to double the RTO on each failure until the segments gets through ("exponential backoff").

Delayed ACK algorithm

A host that is receiving a stream of TCP data segments can increase efficiency in both the Internet and the hosts by sending fewer than one ACK (acknowledgment) segment per data segment received; this is known as a "delayed ACK" [TCP:5].

Delay is max. 500ms.

A delayed ACK gives the application an opportunity to update the window and perhaps to send an immediate response. In particular, in the case of character-mode remote login, a delayed ACK can reduce the number of segments sent by the server by a factor of 3 (ACK, window update, and echo character all combined in one segment).

In addition, on some large multi-user hosts, a delayed ACK can substantially reduce protocol processing overhead by reducing the total number of packets to be processed [TCP:5]. However, excessive delays on ACK's can disturb the round-trip timing and packet "clocking" algorithms [TCP:7].

a TCP receiver SHOULD send an immediate ACK when the incoming segment fills in all or part of a gap in the sequence space.

Nagle's algorithm

RFC896 describes the "small packet problem": when the application sends single-byte messages to the TCP, and it transmitted immediatly in a 41 byte TCP/IP packet (20 bytes IP header, 20 bytes TCP header, 1 byte payload), the result is a 4000% overhead that can cause congestion in the network.

The solution to this problem is to delay the transmission until enough data received from the application and send all collected data in one packet. Nagle proposed that when a TCP connection has outstanding data that has not yet been acknowledged, small segments should not be sent until the outstanding data is acknowledged.

Silly window avoidance

The Silly Window Syndrome (SWS) is described in RFC813. It occurs when a TCP receiver advertises a small window and the TCP sender immediately sends data to fill the window. Let's take the example when the sender process writes a file into the TCP stream in big chunks, while the receiver process reads the bytes one by one. The first few bytes are transmitted as whole segments until the receiver buffer becomes full. Then the application reads one byte, and a window size 1 is offered to the sender. The sender sends a segment with 1 byte payload immediately, the receiver buffer becomes full, and after reading 1 byte, the offered window is 1 byte again. Thus almost the whole file is transmitted in very small segments.

In order to avoid SWS, both sender and receiver must try to avoid this situation. The receiver must not advertise small windows and the sender must not send small segments when only a small window is advertised.

In RFC813 it is offered that

1. the receiver should not advertise windows that is smaller than the maximum segment size of the connection
2. the sender should wait until the window is large enough for a maximum sized segment.

Timestamp option

Efficient retransmissions depends on precious RTT measurements. Packet losses can reduce the precision of these measurements radically. When a segment lost, the ACKs received in that window can not be used; thus reducing the sample rate to one RTT data per window. This is unacceptable if the window is large.

The proposed solution to the problem is to use a separate timestamp field to connect the request and the response on the sender side. The timestamp is transmitted as a TCP option. The option contains two 32-bit timestamps:

Kind=5	Length=10	TS Value	

Here the TS Value (TSVal) field is the current value of the timestamp clock of the TCP sending the option, TS Echo Reply (TSecr) field is 0 or echoes the timestamp value of that was sent by the remote TCP. The TSscr field is valid only in ACK segments that acknowledges new data. Both parties should send the TS option in their SYN segment in order to allow the TS option in data segments.

The timestamp option can also be used for PAWS (protection against wrapped sequence numbers).

14.1.5 Congestion control

Flow control allows the sender to slow down the transmission when the receiver can not accept them because of memory limitations. However there are other situations when a slow down is desirable. If the sender transmits a lot of data into the network it can overload the processing capacities of the network nodes, so packets are lost in the network layer.

For this purpose another window is maintained at the sender side, the congestion window (CWND). The congestion window is a sender-side limit on the amount of data the sender can transmit into the network before receiving ACK. More precisely, the sender can send at most max(CWND, WND) bytes above SND.UNA, therefore $SND.NXT < SND.UNA + max(CWND, WND)$ is guaranteed.

The size of the congestion window is dinamically determined by monitoring the state of the network.

Slow Start and Congestion Avoidance

There are two algorithm that updates the congestion window, "Slow Start" and "Congestion Avoidance". They are specified in RFC2581.

$cwnd \leftarrow 2 * SMSS$
$ssthresh \leftarrow$ `upper bound of the window (e.g. 65536)`
whenever `ACK received`
  if $cwnd < ssthresh$
    $cwnd \leftarrow cwnd + SMSS$
  otherwise
    $cwnd \leftarrow cwnd + SMSS * SMSS/cwnd$
whenever `packet loss detected`
  $cwnd \leftarrow SMSS$
  $ssthresh \leftarrow max(FlightSize/2, 2 * SMSS)$

Slow Start means that when the connection opened the sender initially sends the data with a low rate. This means that the initial window (IW) is at most 2 MSS, but no more than 2 segments. If there was no packet loss, then the congestion window is increased rapidly, it is doubled in each flight. When a packet loss is detected, the congestion window is reset to 1 MSS (loss window, LW) and the "Slow Start" is applied again.

NOTE: RFC3390 increased the IW to roughly 4K bytes: $min(4 * MSS, max(2 * MSS, 4380))$.

When the congestion window reaches a certain limit (slow start threshold), the "Congestion Avoidance" algorithm is applied. During "Congestion Avoidance" the window is incremented by 1 MSS per round-trip-time (RTT). This is usually implemented by updating the window according to the $cwnd += SMSS * SMSS/cwnd$ formula on every non-duplicate ACK.

The Slow Start Threshold is updated when a packet loss is detected. It is set to $max(FlightSize/2, 2 * SMSS)$.

How the sender estimates the flight size? The data sent, but not yet acknowledged.

How the sender detect packet loss? Retransmission timer expired.

Fast Retransmit and Fast Recovery

RFC2581 specifies two additional methods to increase the efficiency of congestion control: "Fast Retransmit" and "Fast Recovery".

"Fast Retransmit" requires that the receiver signal the event, when an out-of-order segment arrives. It is achieved by sending an immediate duplicate ACK. The receiver also sends an immediate ACK when the incoming segment fills in a gap or part of a gap.

When the sender receives the duplicated ACK it knows that some segment after that sequence number is received out-of-order or that the network duplicated the ACK. If 3 duplicated ACK received then it is more likely that a segment was dropped or delayed. In this case the sender starts to retransmit the segments immediately.

"Fast Recovery" means that "Slow Start" is not applied when the loss is detected as 3 duplicate ACKs. The arrival of the duplicate ACKs indicates that the network is not fully congested, segments after the lost segment arrived, as well the ACKs.

Loss Recovery Using Limited Transmit

If there is not enough data to be send after a lost segment, then the Fast Retransmit algorithm is not activated, but the costly retranmission timeout used.

RFC3042 suggests that the sender TCP should send a new data segment in response to each of the first two duplicate acknowledgement. Transmitting these segments increases the probability that TCP can recover from a single lost segment using the fast retransmit algorithm, rather than using a costly retransmission timeout.

Selective Acknowledgments

With selective acknowledgments (SACK), the data receiver can inform the sender about all segments that have arrived successfully, so the sender need retransmit only the segments that have actually been lost.

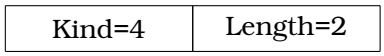
With the help of this information the sender can detect

- replication by the network
- false retransmit due to reordering
- retransmit timeout due to ACK loss

- early retransmit timeout

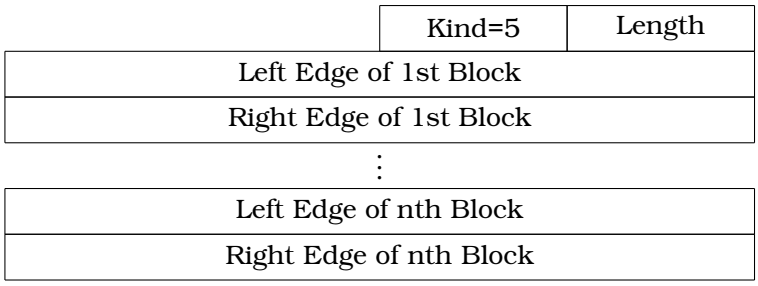
In the congestion control algorithms described so far the sender has only rudimentary information about which segments arrived at the receiver. On the other hand the algorithms are implemented completely on the sender side, they only require that the client sends immediate ACKs on duplicate segments. Therefore they can work in a heterogenous environment, e.g. a client with Tahoe TCP can communicate with a NewReno server. On the other hand SACK must be supported by both endpoint of the connection to be used.

If a TCP supports SACK it includes the *SACK-Permitted* option in the SYN/SYN-ACK segment when initiating the connection. The SACK extension enabled for the connection if the *SACK-Permitted* option was sent and received by both ends. The option occupies 2 octets in the TCP header:



If the SACK is enabled then the data receiver adds SACK option to the ACK segments. The SACK option informs the sender about non-contiguous blocks of data that have been received and queued. The meaning of the *Acknowledgement Number* is unchanged, it is still the cumulative sequence number. Octets received before the *Acknowledgement Number* are kept by the receiver, and can be deleted from the sender's buffer. However the receiver is allowed to drop the segments that was only reported in the SACK option.

The *SACK* option contains the following fields:



Each block represents received bytes of data that are contiguous and isolated with one exception: if a segment received that was already ACKed (i.e. below $RCV.NXT$), it is included as the first block of the *SACK* option. The purpose is to inform the sender about a spurious retransmission.

Each block in the option occupies 8 octets. The TCP header allows 40 bytes for options, so at most 4 blocks can be reported in the *SACK* option (or 3 if TS option is also used). The first block is used for reporting the most recently received data, the following blocks repeats the most recently reported SACK blocks. This way each segment is reported at least 3 times, so the sender receives the information even if some ACK segment is lost.

SACK based loss recovery

Now lets see how the sender can use the information in the *SACK* option. First notice that it can give a better estimation of the amount of data outstanding in the network (called $pipe$ in RFC3517). If $highACK$ is the highest ACKed sequence number, and $highData$ of the highest sequence number transmitted, then the bytes between $highACK$ and $highData$ can be in the network. However $pipe \neq highData - highACK$ if there are lost and retransmitted segments:

$$pipe = highData - highACK - lostBytes + retransmittedBytes$$

A segment is supposed to be lost if it was not received but 3 segments recevied that comes after this segment in the sequence number space. This condition is detected by the sender by receiving either 3 discontiguous SACKed blocks, or at least $3 * SMSS$ SACKed bytes above the sequence numbers of the lost segment.

The transmission of data starts with a *Slow Start* phase. If the loss is detected by 3 duplicate ACK, the sender goes into the recovery state: it sets $cwnd$ and $ssthresh$ to $FlightSize/2$. It also remembers the $highData$ variable, because the recovery state is left when this sequence number is acknowledged.

In the recovery state it sends data until there is space in the congestion window (i.e. $cwnd - pipe >= 1SMSS$) The data of the segment is choosen by the following rules (first rule that applies):

1. send segments that is lost and not yet retransmitted
2. send segments that is not yet transmitted
3. send segments that is not yet retransmitted and possibly fills a gap (there is SACKed data above it)

If there is no data to send, then the sender waits for the next ACK, updates its variables based on the data of the received ACK, and then try to transmit according to the above rules.

If an RTO occurs, the sender drops the collected SACK information and initiates a Slow Start. This is to avoid a deadlock when the receiver dropped a previously SACKed segment.

14.2 TCP module

The `TCP` simple module is the main implementation of the TCP protocol in the INET framework. Other implementation are described in section 14.6. The `TCP` module as other transport protocols work above the network layer and below the application layer, therefore it has gates to be connected with the IPv4 or IPv6 network (ipIn/ipOut or ipv6In/ipv6Out), and with the applications (appIn[k], appOut[k]). One `TCP` module can serve several application modules, and several connections per application. The kth application connects to `TCP`'s `appIn[k]` and `appOut[k]` ports.

The TCP module usually specified by its module interface (`ITCP`) in the NED definition of hosts, so it can be replaced with any implementation that communicates through the same gates. The `TCP` model relies on sending and receiving `IPControlInfo` objects attached to TCP segment objects as control info (see `cMessage::setControlInfo()`).

The `TCP` module manages several `TCPConnection` object each holding the state of one connection. The connections are identified by a connection identifier which is choosen by the application. If the connection is established it can also be identified by the local and remote addresses and ports. The TCP module simply dispatches the incoming application commands and packets to the corresponding object.

14.2.1 TCP packets

The INET framework models the TCP header with the `TCPSegment` message class. This contains the fields of a TCP frame, except:

- *Data Offset*: represented by `cMessage::length()`

- *Reserved*
- *Checksum*: modelled by `cMessage::hasBitError()`
- *Options*: only EOL, NOP, MSS, WS, SACK_PERMITTED, SACK and TS are possible
- *Padding*

The Data field can either be represented by (see `TCPDataTransferMode`):

- encapsulated C++ packet objects,
- raw bytes as a `ByteArray` instance,
- its byte count only,

corresponding to transfer modes OBJECT, BYTESTREAM, BYTECOUNT resp.

14.2.2 TCP commands

The application and the TCP module communicates with each other by sending `cMessage` objects. These messages are specified in the `TCPCommand.msg` file.

The `TCPCommandCode` enumeration defines the message kinds that are sent by the application to the TCP:

- TCP_C_OPEN_ACTIVE: active open
- TCP_C_OPEN_PASSIVE: passive open
- TCP_C_SEND: send data
- TCP_C_CLOSE: no more data to send
- TCP_C_ABORT: abort connection
- TCP_C_STATUS: request status info from TCP

Each command message should have an attached control info of type `TCPCommand`. Some commands (TCP_C_OPEN_xxx, TCP_C_SEND) use subclasses. The `TCPCommand` object has a `connId` field that identifies the connection locally within the application. `connId` is to be chosen by the application in the open command.

When the application receives a message from the TCP, the message kind is set to one of the `TCPStatusInd` values:

- TCP_I_ESTABLISHED: connection established
- TCP_I_CONNECTION_REFUSED: connection refused
- TCP_I_CONNECTION_RESET: connection reset
- TCP_I_TIME_OUT: connection establish timer went off, or max retransmission count reached
- TCP_I_DATA: data packet
- TCP_I_URGENT_DATA: urgent data packet
- TCP_I_PEER_CLOSED: FIN received from remote TCP
- TCP_I_CLOSED: connection closed normally
- TCP_I_STATUS: status info

These messages also have an attached control info with `TCPCommand` or derived type (TCPConnectInfo, TCPStatusInfo, TCPErrorInfo).

14.2.3 TCP parameters

The `TCP` module has the following parameters:

- `advertisedWindow` in bytes, corresponds with the maximal receiver buffer capacity (Note: normally, NIC queues should be at least this size, default is 14*mss)
- `delayedAcksEnabled` delayed ACK algorithm (RFC 1122) enabled/disabled
- `nagleEnabled` Nagle's algorithm (RFC 896) enabled/disabled
- `limitedTransmitEnabled` Limited Transmit algorithm (RFC 3042) enabled/disabled (can be used for TCPReno/TCPTahoe/TCPNewReno/TCPNoCongestionControl)
- `increasedIWEnabled` Increased Initial Window (RFC 3390) enabled/disabled
- `sackSupport` Selective Acknowledgment (RFC 2018, 2883, 3517) support (header option) (SACK will be enabled for a connection if both endpoints support it)
- `windowScalingSupport` Window Scale (RFC 1323) support (header option) (WS will be enabled for a connection if both endpoints support it)
- `timestampSupport` Timestamps (RFC 1323) support (header option) (TS will be enabled for a connection if both endpoints support it)
- `mss` Maximum Segment Size (RFC 793) (header option, default is 536)
- `tcpAlgorithmClass` the name of TCP flavour

  Possible values are "TCPReno" (default), "TCPNewReno", "TCPTahoe", "TCPNoCongestionControl" and "DumpTCP". In the future, other classes can be written which implement Vegas, LinuxTCP or other variants. See section 14.4 for detailed description of implemented flavours.

  Note that TCPOpenCommand allows tcpAlgorithmClass to be chosen per-connection.

- `recordStats` if set to false it disables writing excessive amount of output vectors

14.2.4 Statistics

The `TCP` module collects the following vectors:

`send window`	$SND.WND$
`receive window`	$RCV.WND$, after SWS avoidance applied
`advertised window`	$RCV.NXT + RCV.WND$
`sent seq`	*Sequence Number* of the sent segment
`sent ack`	*Acknowledgement Number* of the sent segment
`rcvd seq`	*Sequence Number* of the received segment
`rcvd ack`	*Acknowledgement Number* of the received segment
`unacked bytes`	number of sent and unacknowledged bytes ($maxofSND.NXT - SND.UNA$)
`rcvd dupAcks`	number of duplicate acknowledgements, reset to 0 when $SND.UNA$ advances
`pipe`	the value of the SACK $pipe$ variable (estimated number of bytes outstanding in the network)
`sent sacks`	number of SACK blocks sent
`rcvd sacks`	number of SACK blocks received
`rcvd oooseg`	number of received out-of-order segments
`rcvd naseg`	number of received unacceptable segments (outside the receive window)
`rcvd sackedBytes`	total amount of SACKed bytes in the buffer of the sender
`tcpRcvQueueBytes`	number of bytes in the receiver's buffer
`tcpRcvQueueDrops`	number of bytes dropped by the receiver (not enough buffer)
`cwnd`	congestion window
`ssthresh`	slow start threshold
`measured RTT`	measured round trip time
`smoothed RTT`	smoothed round trip time
`RTTVAR`	measured smoothed variance of round trip time
`RTO`	retransmission timeout
`numRTOs`	number of retransmission timeouts occured

If the `recordStats` parameter is set to **`false`**, then none of these output vectors are generated.

14.3 TCP connections

Most part of the TCP specification is implemented in the `TCPConnection` class: takes care of the state machine, stores the state variables (TCB), sends/receives SYN, FIN, RST, ACKs, etc. TCPConnection itself implements the basic TCP "machinery", the details of congestion control are factored out to `TCPAlgorithm` classes.

There are two additional objects the `TCPConnection` relies on internally: instances of `TCPSendQueue` and `TCPReceiveQueue`. These polymorph classes manage the actual data stream, so `TCPConnection` itself only works with sequence number variables. This makes it possible to easily accomodate need for various types of simulated data transfer: real byte stream, "virtual" bytes (byte counts only), and sequence of `cMessage` objects (where every message object is mapped to a TCP sequence number range).

14.3.1 Data transfer modes

Different applications have different needs how to represent the messages they communicate with. Sometimes it is enough to simulate the amount of data transmitted ("200 MB"), contents does not matter. In other scenarios contents matters a lot. The messages can be represented

as a stream of bytes, but sometimes it is easier for the applications to pass message objects to each other (e.g. HTTP request represented by a `HTTPRequest` message class).

The TCP modules in the INET framework support 3 data transfer modes:

- `TCP_TRANSFER_BYTECOUNT`: only byte counts are represented, no actual payload in `TCPSegment`s. The TCP sends as many TCP segments as needed
- `TCP_TRANSFER_BYTESTREAM`: the application can pass byte arrays to the TCP. The sending TCP breaks down the bytes into MSS sized chunks and transmits them as the payload of the TCP segments. The receiving application can read the chunks of the data.
- `TCP_TRANSFER_OBJECT`: the application pass a `cMessage` object to the TCP. The sending TCP sends as many TCP segments as needed according to the message length. The `cMessage` object is also passed as the payload of the first segment. The receiving application receives the object only when its last byte is received.

These values are defined in `TCPCommand.msg` as the `TCPDataTransferMode` enumeration. The application can set the data transfer mode per connection when the connection is opened. The client and the server application must specify the same data transfer mode.

14.3.2 Opening connections

Applications can open a local port for incoming connections by sending the TCP a TCP_C_PASSIVE_OPEN message. The attached control info (an `TCPOpenCommand`) contains the local address and port. The application can specify that it wants to handle only one connection at a time, or multiple simultanous connections. If the `fork` field is true, it emulates the Unix accept(2) semantics: a new connection structure is created for the connection (with a new `connId`), and the connection with the old connection id remains listening. If `fork` is false, then the first connection is accepted (with the original `connId`), and further incoming connections will be refused by the TCP by sending an RST segment. The `dataTransferMode` field in `TCPOpenCommand` specifies whether the application data is transmitted as C++ objects, real bytes or byte counts only. The congestion control algorithm can also be specified on a per connection basis by setting `tcpAlgorithmClass` field to the name of the algorithm.

The application opens a connection to a remote server by sending the TCP a TCP_C_OPEN_ACTIVE command. The TCP creates a `TCPConnection` object an sends a SYN segment. The initial sequence number selected according to the simulation time: 0 at time 0, and increased by 1 in each 4μs. If there is no response to the SYN segment, it retry after 3s, 9s, 21s and 45s. After 75s a connection establishment timeout (TCP_I_TIMEOUT) reported to the application and the connection is closed.

When the connection gets established, TCP sends a TCP_I_ESTABLISHED notification to the application. The attached control info (a `TCPConnectInfo` instance) will contain the local and remote addresses and ports of the connection. If the connection is refused by the remote peer (e.g. the port is not open), then the application receives a TCP_I_CONNECTION_REFUSED message.

NOTE: If you do active OPEN, then send data and close before the connection has reached ESTABLISHED, the connection will go from SYN_SENT to CLOSED without actually sending the buffered data. This is consistent with RFC 793 but may not be what you would expect.

NOTE: Handling segments with SYN+FIN bits set (esp. with data too) is inconsistent across TCPs, so check this one if it is of importance.

14.3.3 Sending Data

The application can write data into the connection by sending a message with TCP_C_SEND kind to the TCP. The attached control info must be of type `TCPSendCommand`.

The TCP will add the message to the *send queue*. There are three type of send queues corresponding to the three data transfer mode. If the payload is transmitted as a message object, then `TCPMsgBasedSendQueue`; if the payload is a byte array then `TCPDataStreamSendQueue`; if only the message lengths are represented then `TCPVirtualDataSendQueue` are the classes of send queues. The appropriate queue is created based on the value of the `dataTransferMode` parameter of the Open command, no further configuration is needed.

The message is handed over to the IP when there is enough room in the windows. If Nagle's algorithm is enabled, the TCP will collect 1 SMSS data and sends them toghether.

NOTE: There is no way to set the PUSH and URGENT flags, when sending data.

14.3.4 Receiving Data

The TCP connection stores the incoming segments in the *receive queue*. The receive queue also has three flavours: `TCPMsgBasedRcvQueue`, `TCPDataStreamRcvQueue` and `TCPVirtualDataRcvQueue`. The queue is created when the connection is opened according to the `dataTransferMode` of the connection.

Finite receive buffer size is modeled by the `advertisedWindow` parameter. If receive buffer is exhausted (by out-of-order segments) and the payload length of a new received segment is higher than the free receiver buffer, the new segment will be dropped. Such drops are recorded in *tcpRcvQueueDrops* vector.

If the *Sequence Number* of the received segment is the next expected one, then the data is passed to the application immediately. The `recv()` call of Unix is not modeled.

The data of the segment, which can be either a `cMessage` object, a `ByteArray` object, or a simply byte count, is passed to the application in a message that has TCP_I_DATA kind.

NOTE: The TCP module does not handle the segments with PUSH or URGENT flags specially. The data of the segment passed to the application as soon as possible, but the application can not find out if that data is urgent or pushed.

14.3.5 RESET handling

When an error occures at the TCP level, an RST segment is sent to the communication partner and the connection is aborted. Such error can be:

- arrival of a segment in CLOSED state
- an incoming segment acknowledges something not yet sent.

The receiver of the RST it will abort the connection. If the connection is not yet established, then the passive end will go back to the LISTEN state and waits for another incoming connection instead of aborting.

14.3.6 Closing connections

When the application does not have more data to send, it closes the connection by sending a TCP_C_CLOSE command to the TCP. The TCP will transmit all data from its buffer and in the last segment sets the FIN flag. If the FIN is not acknowledged in time it will be retransmitted with exponential backoff.

The TCP receiving a FIN segment will notify the application that there is no more data from the communication partner. It sends a TCP_I_PEER_CLOSED message to the application containing the connection identifier in the control info.

When both parties have closed the connection, the applications receive a TCP_I_CLOSED message and the connection object is deleted. (Actually one of the TCPs waits for $2MSL$ before deleting the connection, so it is not possible to reconnect with the same addresses and port numbers immediately.)

14.3.7 Aborting connections

The application can also abort the connection. This means that it does not wait for incoming data, but drops the data associated with the connection immediately. For this purpose the application sends a TCP_C_ABORT message specifying the connection identifier in the attached control info. The TCP will send a RST to the communication partner and deletes the connection object. The application should not reconnect with the same local and remote addresses and ports within MSL (maximum segment lifetime), because segments from the old connection might be accepted in the new one.

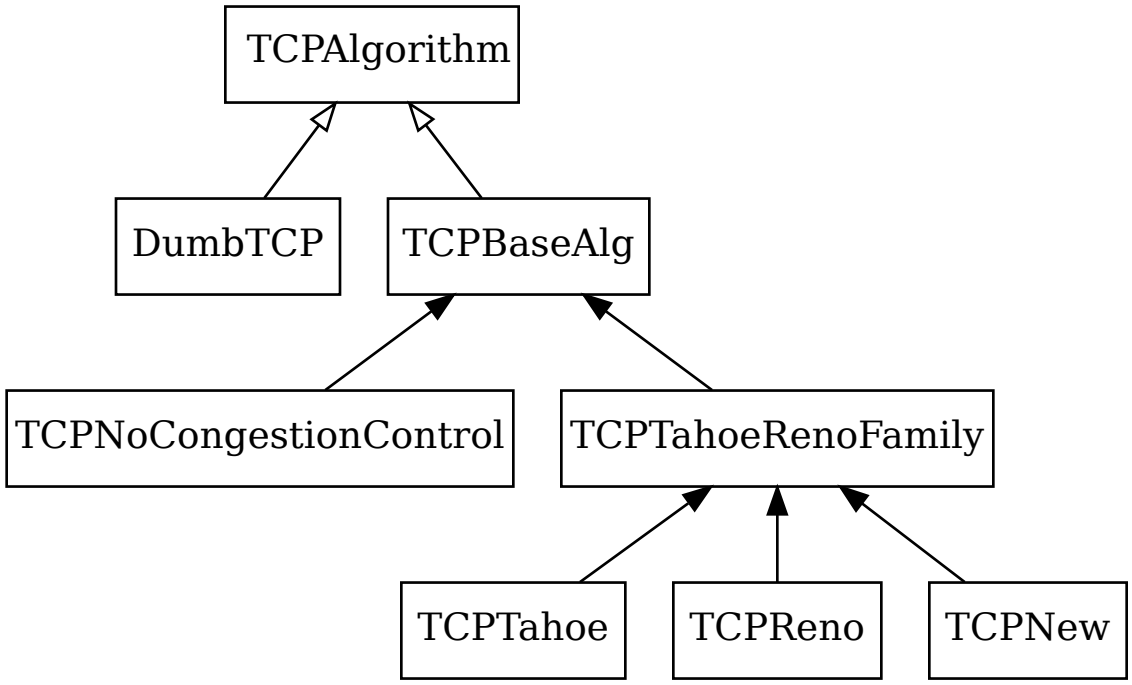
14.3.8 Status Requests

Applications can get detailed status information about an existing connection. For this purpose they send the TCP module a TCP_C_STATUS message attaching an `TCPCommand` info with the identifier of the connection. The TCP will respond with a TCP_I_STATUS message with a `TCPStatusInfo` attachement. This control info contains the current state, local and remote addresses and ports, the initial sequence numbers, windows of the receiver and sender, etc.

14.4 TCP algorithms

The `TCPAlgorithm` object controls retransmissions, congestion control and ACK sending: delayed acks, slow start, fast retransmit, etc. They are all extends the `TCPAlgorithm` class. This simplifies the design of `TCPConnection` and makes it a lot easier to implement TCP variations such as Tahoe, NewReno, Vegas or LinuxTCP.

Currently implemented algorithm classes are `TCPReno`, `TCPTahoe`, `TCPNewReno`, `TCPNoCongestionControl` and `DumbTCP`. It is also possible to add new TCP variations by implementing `TCPAlgorithm`.



The concrete TCP algorithm class to use can be chosen per connection (in OPEN) or in a module parameter.

14.4.1 DumbTCP

A very-very basic `TCPAlgorithm` implementation, with hardcoded retransmission timeout (2 seconds) and no other sophistication. It can be used to demonstrate what happened if there was no adaptive timeout calculation, delayed acks, silly window avoidance, congestion control, etc. Because this algorithm does not send duplicate ACKs when receives out-of-order segments, it does not work well together with other algorithms.

14.4.2 TCPBaseAlg

The `TCPBaseAlg` is the base class of the INET implementation of Tahoe, Reno and New Reno. It implements basic TCP algorithms for adaptive retransmissions, persistence timers, delayed ACKs, Nagle's algorithm, Increased Initial Window – EXCLUDING congestion control. Congestion control is implemented in subclasses.

Delayed ACK

When the `delayedAcksEnabled` parameter is set to **`true`**, `TCPBaseAlg` applies a 200ms delay before sending ACKs.

Nagle's algorithm

When the `nagleEnabled` parameter is **`true`**, then the algorithm does not send small segments if there is outstanding data. See also 14.1.4.

Persistence Timer

The algorithm implements *Persistence Timer* (see 14.1.3). When a zero-sized window is received it starts the timer with 5s timeout. If the timer expires before the window is increased, a 1-byte probe is sent. Further probes are sent after 5, 6, 12, 24, 48, 60, 60, 60, ... seconds until the window becomes positive.

Initial Congestion Window

Congestion window is set to 1 SMSS when the connection is established. If the `increasedIWEnabled` parameter is true, then the initial window is increased to 4380 bytes, but at least 2 SMSS and at most 4 SMSS. The congestion window is not updated afterwards; subclasses can add congestion control by redefining virtual methods of the `TCPBaseAlg` class.

Duplicate ACKs

The algorithm sends a duplicate ACK when an out-of-order segment is received or when the incoming segment fills in all or part of a gap in the sequence space.

RTO calculation

Retransmission timeout (RTO) is calculated according to Jacobson algorithm (with $\alpha = 7/8$), and Karn's modification is also applied. The initial value of the RTO is 3s, its minimum is 1s, maximum is 240s (2 MSL).

14.4.3 TCPNoCongestion

TCP with no congestion control (i.e. congestion window kept very large). Can be used to demonstrate effect of lack of congestion control.

14.4.4 TCPTahoe

The `TCPTahoe` algorithm class extends `TCPBaseAlg` with *Slow Start*, *Congestion Avoidance* and *Fast Retransmit* congestion control algorithms. This algorithm initiates a *Slow Start* when a packet loss is detected.

Slow Start

The congestion window is initially set to 1 SMSS or in case of `increasedIWEnabled` is **`true`** to 4380 bytes (but no less than 2 SMSS and no more than 4 SMSS). The window is increased on each incoming ACK by 1 SMSS, so it is approximately doubled in each RTT.

Congestion Avoidance

When the congestion window exceeded $ssthresh$, the window is increased by $SMSS^2/cwnd$ on each incoming ACK event, so it is approximately increased by 1 SMSS per RTT.

Fast Retransmit

When the 3rd duplicate ACK received, a packet loss is detected and the packet is retransmitted immediately. Simultanously the $ssthresh$ variable is set to half of the $cwnd$ (but at least 2 SMSS) and $cwnd$ is set to 1 SMSS, so it enters slow start again.

Retransmission timeouts are handled the same way: $ssthresh$ will be $cwnd/2$, $cwnd$ will be 1 SMSS.

14.4.5 TCPReno

The `TCPReno` algorithm extends the behaviour `TCPTahoe` with *Fast Recovery*. This algorithm can also use the information transmitted in SACK options, which enables a much more accurate congestion control.

Fast Recovery

When a packet loss is detected by receiveing 3 duplicate ACKs, $ssthresh$ set to half of the current window as in Tahoe. However $cwnd$ is set to $ssthresh + 3 * SMSS$ so it remains in congestion avoidance mode. Then it will send one new segment for each incoming duplicate ACK trying to keep the pipe full of data. This requires the congestion window to be inflated on each incoming duplicate ACK; it will be deflated to $ssthresh$ when new data gets acknowledged.

However a hard packet loss (i.e. RTO events) cause a slow start by setting $cwnd$ to 1 SMSS.

SACK congestion control

This algorithm can be used with the SACK extension. Set the `sackSupport` parameter to **`true`** to enable sending and receiving *SACK* options.

14.4.6 TCPNewReno

This class implements the TCP variant known as New Reno. New Reno recovers more efficiently from multiple packet losses within one RTT than Reno does.

It does not exit fast-recovery phase until all data which was out-standing at the time it entered fast-recovery is acknowledged. Thus avoids reducing the $cwnd$ multiple times.

14.5 TCP socket

`TCPSocket` is a convenience class, to make it easier to manage TCP connections from your application models. You'd have one (or more) `TCPSocket` object(s) in your application simple module class, and call its member functions (bind(), listen(), connect(), etc.) to open, close or abort a TCP connection.

TCPSocket chooses and remembers the connId for you, assembles and sends command packets (such as OPEN_ACTIVE, OPEN_PASSIVE, CLOSE, ABORT, etc.) to TCP, and can also help you deal with packets and notification messages arriving from TCP.

A session which opens a connection from local port 1000 to 10.0.0.2:2000, sends 16K of data and closes the connection may be as simple as this (the code can be placed in your `handleMessage()` or `activity()`):

```
TCPSocket socket;
socket.connect(IPvXAddress("10.0.0.2"), 2000);

msg = new cMessage("data1");
msg->setByteLength(16*1024);  16K
socket.send(msg);

socket.close();
```

Dealing with packets and notification messages coming from TCP is somewhat more cumbersome. Basically you have two choices: you either process those messages yourself, or let TCPSocket do part of the job. For the latter, you give TCPSocket a callback object on which it'll invoke the appropriate member functions: `socketEstablished()`, `socketDataArrived()`, `socketFailure()`, `socketPeerClosed()`, etc (these are methods of `TCPSocket::CallbackInterface`)., The callback object can be your simple module class too.

This code skeleton example shows how to set up a TCPSocket to use the module itself as callback object:

```
class MyModule : public cSimpleModule, public TCPSocket::CallbackInterface
{
    TCPSocket socket;
    virtual void socketDataArrived(int connId, void *yourPtr,
                                   cPacket *msg, bool urgent);
    virtual void socketFailure(int connId, void *yourPtr, int code);
    ...
};

void MyModule::initialize() {
    socket.setCallbackObject(this,NULL);
}

void MyModule::handleMessage(cMessage *msg) {
    if (socket.belongsToSocket(msg))
        socket.processMessage(msg); dispatch to socketXXXX() methods
    else
        ...
}

void MyModule::socketDataArrived(int, void *, cPacket *msg, bool) {
    ev << "Received TCP data, " << msg->getByteLength() << " bytes\\n";
    delete msg;
}

void MyModule::socketFailure(int, void *, int code) {
    if (code==TCP_I_CONNECTION_RESET)
        ev << "Connection reset!\\n";
    else if (code==TCP_I_CONNECTION_REFUSED)
        ev << "Connection refused!\\n";
    else if (code==TCP_I_TIMEOUT)
```

```
        ev << "Connection timed out!\\n";
}
```

If you need to manage a large number of sockets (e.g. in a server application which handles multiple incoming connections), the `TCPSocketMap` class may be useful. The following code fragment to handle incoming connections is from the LDP module:

```
TCPSocket *socket = socketMap.findSocketFor(msg);
if (!socket)
{
    not yet in socketMap, must be new incoming connection: add to socketMap
    socket = new TCPSocket(msg);
    socket->setOutputGate(gate("tcpOut"));
    socket->setCallbackObject(this, NULL);
    socketMap.addSocket(socket);
}
dispatch to socketEstablished(), socketDataArrived(), socketPeerClosed()
or socketFailure()
socket->processMessage(msg);
```

14.6 Other TCP implementations

INET contains two other implementation of the TCP protocol: `TCP_lwIP` and `TCP_NSC`. All TCP modules implements the `ITCP` interface and communicate with the application and the IP layer through the same interface. Therefore they can be interchanged and can operate with each other. See `examples/inet/tcpclientserver/omnetpp.ini` how to parametrize `StandardHost`s to use the different implementations.

14.6.1 TCP LWIP

lwIP is a light-weight implementation of the TCP/IP protocol suite that was 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 `TCP_lwIP` 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.

Statistics

The `TCP_lwIP` module generates the following vector files:

- `send window`: value of the $SND.WND$ variable
- `sent seq`: *Sequence Number* of the sent segment
- `sent ack`: *Acknowledgment Number* of the sent segment
- `receive window`: value of the $RCV.WND$ variable
- `rcvd seq`: *Sequence Number* of the received segment
- `rcvd acq`: *Acknowledgment Number* of the received segment

The creation of these vectors can be disabled by setting the `recordStats` parameter to **`false`**.

14.6.2 TCP NSC

Network Simulation Cradle (NSC) is a tool that allow 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, altough on 64-bit systems only Linux implementations can be built.

To use the `TCP_NSC` 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 `TCP_NSC` 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 `TCP_NSC` 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 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.

Statistics

The `TCP_NSC` module collects the following vectors:

- `sent seq` *Sequence Number* of the sent TCP segment
- `sent ack` *Acknowledgment Number* of the sent TCP segment
- `rcvd seq` *Sequence Number* of the received TCP segment
- `rcvd ack` *Acknowledgement Number* of the received TCP segment

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 instance in this table, so you can not create more then 50 instance of `TCP_NSC`. You can increase the `NUM_STACKS` constant in `num_stacks.h` and recompile NSC to overcome this limitation.
- The `TCP_NSC` module does not supprt 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 current window of the connection.

14.7 TCP applications

This sections describes the applications using the TCP protocol. Each application must implement the `ITCPApp` module interface to ease configuring the `StandardHost` module.

The applications described here are all contained by the `inet.applications.tcpapp` package. 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 (behaviour specific to the modelled 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.

14.7.1 TCPBasicClientApp

Client for a generic request-response style protocol over TCP. 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 `TCPGenericSrvApp`; the model sends `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
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.

14.7.2 TCPSinkApp

Accepts any number of incoming TCP connections, and discards whatever arrives on them.

The module parameter `dataTransferMode` should be set the transfer mode in TCP layer. Its possible values ("bytecount", "object", "bytestream") are described in ...

14.7.3 TCPGenericSrvApp

Generic server application for modelling TCP-based request-reply style protocols or applications.

Requires message object preserving sendQueue/receiveQueue classes to be used with `TCP` (that is, TCPMsgBasedSendQueue and TCPMsgBasedRcvQueue; TCPVirtualBytesSendQueue/RcvQueue are not good).

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). `TCPGenericSrvApp` 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).

14.7.4 TCPEchoApp

The `TCPEchoApp` application accepts any number of incoming TCP connections, and sends back the messages that arrive on them, The lengths of the messages are multiplied by `echoFactor` before sending them back (echoFactor=1 will result in sending back the same

message unmodified.) The reply can also be delayed by a constant time (`echoDelay` parameter).

When `TCPEchoApp` receives data packets from TCP (and such, when they can be echoed) depends on the dataTransferMode setting. With "bytecount" and "bytestream", TCP passes up data to us as soon as a segment arrives, so it can be echoed immediately. With "object" mode, our local TCP reproduces the same messages that the sender app passed down to its TCP – so if the sender app sent a single 100 MB message, it will be echoed only when all 100 megabytes have arrived.

14.7.5 TCPSessionApp

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. May act either as client or as server, and works with TCPVirtualBytesSendQueue/RcvQueue as sendQueue/receiveQueue setting for TCP. Compatible with both IPv4 (IPv4) and IPv6.

Opening the connection

Regarding the type of opening the connection, the application may be either a client or a server. When active=false, the application will listen on the given local localPort, and wait for an incoming connection. When active=true, the application will bind to given local localAddress:localPort, and connect to the connectAddress:connectPort. To use an ephemeral port as local port, set the localPort parameter to -1.

Even when in server mode (active=false), 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. With sendScript, the format 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.

14.7.6 TelnetApp

Models Telnet sessions with a specific user behaviour. The server app should be `TCPGenericSrvApp`.

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

Each parameter in the above description is "volatile", so you can use distributions to emulate random behaviour.

Additional parameters: addresses,ports dataTransferMode

> **NOTE:** This module emulates a very specific user behaviour, 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.

14.7.7 TCPSrvHostApp

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

Server threads should be subclassed from the `TCPServerThreadBase` C++ class, registered in the C++ code using the Register_Class() macro, and the class name should be specified in the serverThreadClass parameter of `TCPSrvHostApp`. The thread object will receive events via a callback interface (methods like established(), dataArrived(), peerClosed(), timerExpired()), and can send packets via TCPSocket's send() method.

Example server thread class: `TCPGenericSrvThread`.

> **IMPORTANT:** Before you try to use this module, make sure you actually need it! In most cases, `TCPGenericSrvApp` and `GenericAppMsg` will be completely enough, and they are a lot easier to handle. You'll want to subclass your client from `TCPGenericCliAppBase` then; check `TelnetApp` and `TCPBasicClientApp` for examples.

Chapter 15

The SCTP Model

15.1 Overview

Blah blah blah

Chapter 16

Internet Routing

16.1 Overview

Blah blah blah

Chapter 17

Differentiated Services

17.1 Overview

In the early days of the Internet, only best effort service was defined. The Internet delivers individually each packet, 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 was 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 the 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. At the hops from the source to the destination each router must examine the packets, and decide if it belongs to a reserved application flow. This could cause a memory and processing demand in the routers. Other drawback is that the reservation must be periodically refreshed, so there is an overhead during the data transmission too.

Differentiated Services is a more scalable approach to offer 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 of the domain are allocated to forwarding classes, taking into account the available resources and the traffic flows. There are *service level agggreements* (SLA) 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 protect the domain from misbehaving traffic sources. Nonconforming traffic may be dropped, delayed, or marked with a different forwarding class.

17.1.1 Implemented Standards

The implementation follows these RFCs below:

- 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

17.2 Architecture of NICs

Network Interface Card (NIC) modules, such as `PPPInterface` and `EthernetInterface`, may contain traffic conditioners in their input and output data path. 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 them other way, for differentiated handling in the queues.

The NICs may also contain an external queue component. If the `queueType` parameter is set, it must contain a module type implementing the `IOutputQueue` module interface. If it is not specified, then `PPP` and `EtherMAC` use an internal drop tail queue to buffer the packets until the line is busy.

17.2.1 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 them other way, for differentiated handling in the queues.

Traffic conditioners perform the following actions:

- classify the incoming packets
- meter the traffic in each class
- marks/drops 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.

17.2.2 Output Queues

The queue component also has one input and one output gate. These components must implement a passive queue behaviour: they only deliver a packet, when the module connected to its output explicitly asks them. In terms of C++ it means, that the simple module owning the `out` gate, or which is connected to the `out` gate of the compound module, must implement the `IPassiveQueue` interface. The next module asks a packet by calling the `requestPacket()` method of this interface.

17.3 Simple modules

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

The type of the components are:

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

17.3.1 Queues

When packets arrive at higher rate, than the interface can trasmit, they are getting 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 asks a packet by calling the `requestPacket()` method.

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

FIFO Queue

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

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

DropTailQueue

The other primitive queue module is `DropTailQueue`. Its capacity can be specified by the `frameCapacity` parameter. When the number of stored packet reached 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 `FIFOQueue` can be. However its output can be connected to schedulers.

This module implements the `IOutputQueue` interface, so it can be used as the queue component of interface card per se.

17.3.2 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).

Other kind of droppers are absolute droppers; they drop each received packet. They 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 `ThresholdDropper` and `REDDropper`. These modules has 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.

Threshold Dropper

The `ThresholdDropper` module selectively drops packets, based on the available buffer space of the queues attached to its output. The buffer space can be specified as the count of packets, or as the size in bytes.

The module sums the buffer lengths of its outputs and if enqueuing a packet would exceed the configured capacities, then the packet will be dropped instead.

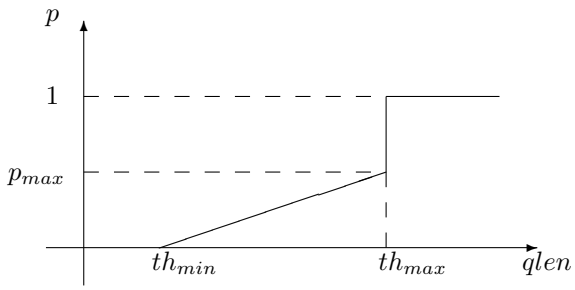
By attaching a `ThresholdDropper` to the input of a FIFO queue, you can compose a drop tail queue. Shared buffer space can be modeled by attaching more FIFO queues to the output.

RED Dropper

The `REDDropper` module implements Random Early Detection ([FJ93]).

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 `IQueueAccess` C++ interface (e.g. `FIFOQueue`).

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.

17.3.3 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 `IPassiveQueue` C++ interface. Schedulers also implement `IPassiveQueue`, so they can be cascaded to other schedulers, and can be used as the output module of `IOutputQueue`s.

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

Priority Scheduler

The `PriorityScheduler` module implements a strict priority scheduler. Packets that arrived on `in[0]` has the highest priority, then packets arrived on `in[1]`, and so on. If more packets available when one is requested, then the one with 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 packets can starve lower priority packets. Therefore it is necessary to limit the rate of higher priority packets to a fraction of the output datarate.

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

If the size of the packets are equal, then `WRRScheduler` divides the available bandwith according to the weights. In each case, it allocates the bandwith fairly. Each flow receives a guaranteed minimum bandwith, which is ensured even if other flows exceed their share (flow isolation). It is 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.

17.3.4 Classifiers

Classifier modules have one input and many output gates. They examine the received packets, and forward them to the appropriate output gate based on the content of some portion of the packet header. You can read more about classifiers in RFC 2475 2.3.1 and RFC 3290 4.

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, protocol, source/destination port range, and bits of TypeOfService/TrafficClass field to be matched. They also specify the index of the output gate matching packet should be forwarded to. 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. The filter element has a mandatory `@gate` attribute that gives the index of the gate for packets matching the filter. Other attributes are optional and specify the condition of matching:

- `@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,

- 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]`, so on. If the received packet has a DSCP value not enumerated in the `dscps` parameter, it will be forwarded to the `defaultOut` gate.

17.3.5 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 decided to be green (in-profile) or red (out-of-profile). Some meters apply more than two conformance level, e.g. in three color meters the partially conforming packets are classified as yellow.

The allowed traffic profile is usually specified by a token bucket. In this model, a bucket is filled in with tokens with 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 that tokens are removed, and the packet marked as conforming to the traffic profile. If the bucket contains less tokens than needed, it left unchanged, but the packet marked as non-conforming.

Meters has 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 can not be changed to a less conforming color: if a packet is classified as non-conforming by a meter, it also handled as non-conforming in later 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 which 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 output, 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), Committed Burst Size (cbs), to be either green, or red.

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

SingleRateThreeColorMeter

The `SingleRateThreeColorMeter` module implements a Single Rate Three Color Meter (RFC 2697). The module has three output for 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), to be either green, yellow or red. The green traffic is guaranteed to be under $cir * (t_1 - t_0) + 8 * cbs$, while the green+yellow traffic to be under $cir * (t_1 - t_0) + 8 * (cbs + ebs)$ 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 the green, yellow, and red packets.

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

17.3.6 Markers

DSCP markers sets the codepoint of the crossing 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 bit of TypeOfService/TrafficClass) of IP datagrams to the value specified by the `dscps` parameter. The `dscps` parameter is a space separated list of codepoints. You can specify a different value for each input gate; packets arrived at the i^{th} input gate are marked with the i^{th} value. If there are fewer values, than gates, then the last one is used for extra gates.

The DSCP values are enumerated in the `DSCP.msg` file. You can use both names and integer values in the `dscps` parameter.

For example the following lines are equivalent:

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

17.4 Compound modules

17.4.1 AFxyQueue

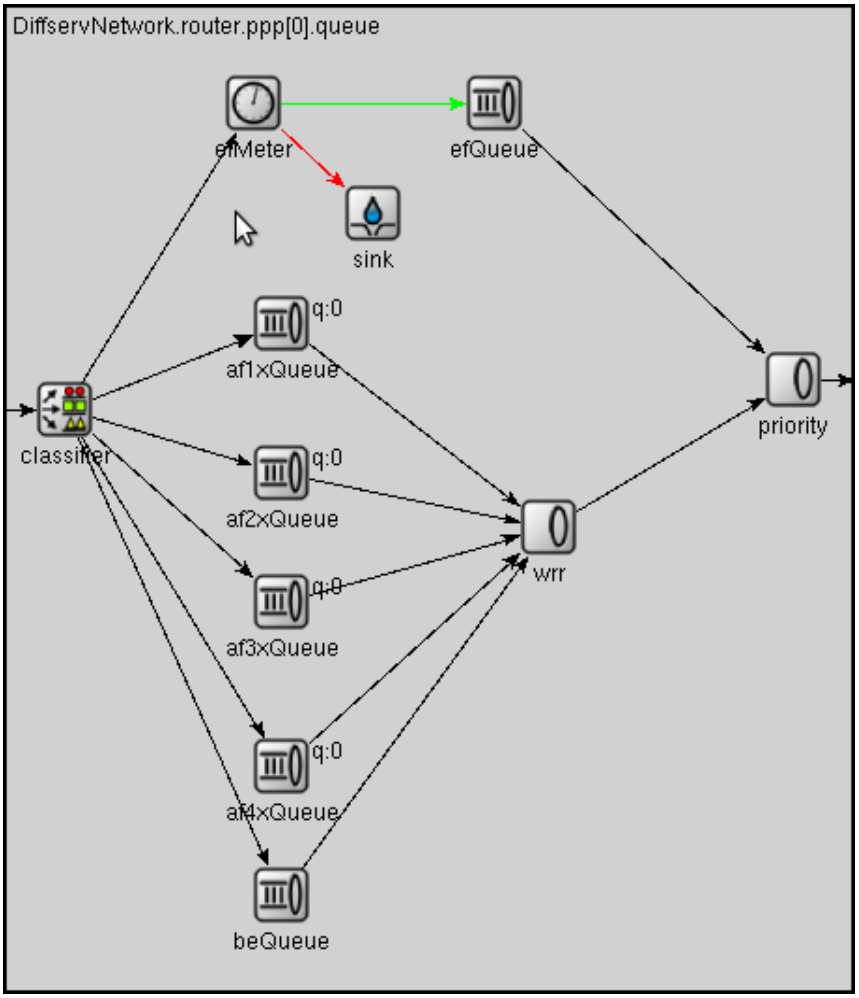
The `AFxyQueue` module is an example queue, that implements one class of the Assured Forwarding PHB group (RFC 2597).

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.

17.4.2 DiffservQeueue

The `DiffservQueue` is an example queue, that can be used in interfaces of DS 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. DSCPs other than AFxy and EF are handled as BE (best effort).

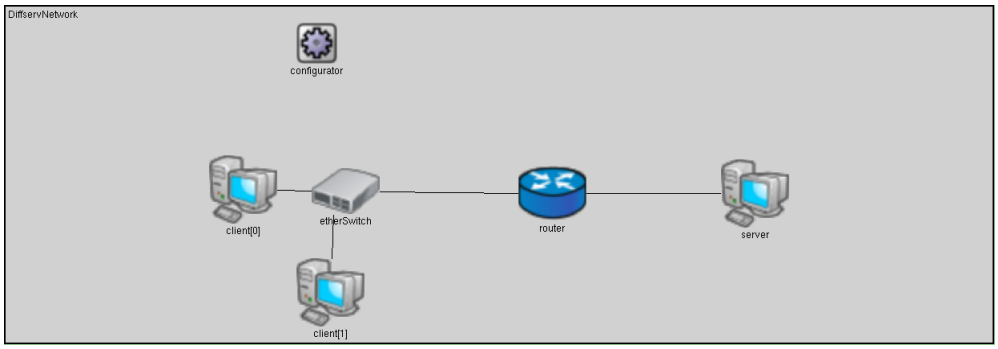
EF packets are stored in a dedicated queue, and served first when a packet is requested. Because they can preempt the other queues, the rate of the EF packets should be limited to a fraction of the bandwith of the link. This is achieved by metering the EF traffic with a token bucket meter and dropping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

There are other queues for AFx classes and BE. The AFx queues use RED to implement 3 different drop priorities within the class. BE packets are stored in a drop tail queue. Packets from AFxy and BE queues are sheduled by a WRR scheduler, which ensures that the remaining bandwith is allocated among the classes according to the specified weights.

17.5 Examples

17.5.1 Simple domain example

The `examples/diffserv/simple_` directory contains a simple demonstration of Diffserv QoS capabilities.



The network contains a router with an 10Mbps Ethernet interface and a 128kbps dialup link to a server. Clients are configured to generate high traffic, that causes congestion at the router PPP interface. One of the clients sends VoIP packets to the server.

With the `VoIP_WithoutQoS` configuration, the queue in the router PPP interface will be overloaded and packets are dropped randomly.

With the `VoIP_WithPolicing` configuration, the VoIP traffic is classified as EF traffic and the necessary bandwidth is allocated for it. The traffic conditioning is configured so, that packets generated by the other clients are dropped if they exceed the enabled rate.

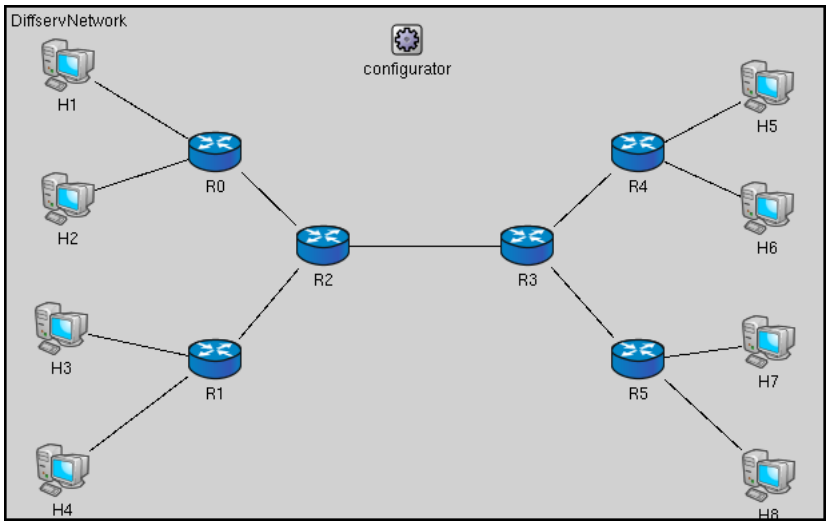
With the `VoIP_WithPolicingAndQueuing` configuration, the VoIP traffic is classified as EF traffic and the necessary bandwidth is allocated for it. The router's queue is configured so that EF packets are prioritized over other packets, so lower delays are expected.

After running these configuration you can see the statistics by opening the VoIP.anf file, or can hear the difference by comparing the recorded .wav files in the results directory.

17.5.2 One domain example

The `examples/diffserv/onedomain` directory contains a network in which the routers constitute a single DiffServ domain. Several experiments are performed on this network, that demonstrates the resource allocation, and flow isolation capabilities of DiffServ network compared to the Best Effort service. It also shows the behavior of the different forwarding classes. These experiments reproduce the results obtained by an NS3 DiffServ implementation ([Ram10]).

The network composed of 8 hosts and 6 routers. Hosts are connected to the routers by 100Mbps Ethernet links. The routers are connected by PPP lines with lower datarates. Traffic policing performed at the Ethernet interface of the edge routers (`R0`,`R1`,`R4`,`R5`). By changing the queue module of the PPP interfaces, diverse per-hop behaviors can be instrumented.



Traffic flows from hosts `H1–H4` to hosts `H5–H8`. The source hosts configured to send voice and video data to the destinations hosts. The voice is sent by an `UDPBacisBurst` application, with mean burst duration 352ms, and mean sleep duration 650ms; this mirrors the talk/listen phases of a phone call. 172 bytes long packets (160 byte data + 12 byte RTP header) are sent in every 20ms in the burst phase. Considering the IP+UDP overhead, this results a 80kbps peek rate, and approximately 28kbps avarege rate for the voice traffic. The video streaming application is modeled by a 100kbps CBR traffic, 500 bytes are sent in every 40ms by an `UDPBasicApp` application.

Traffic policing performed at the Ethernet interfaces of the edge routers (`R0`,`R1`,`R4`,`R5`). For this purpose a traffic conditioner is added to the ingress data path of the Ethernet interfaces. The necessary forwarding behaviours are implemented by changing the external queue module of the PPP interfaces of routers.

The following sections describe four experiments in detail. The experiments are numbered from 1 to 5, but 4 is omitted, to keep it consistent with the numbering in [Ram10]. The ini file define the configurations for each experiment. Configurations named `ExpXY` (where X and Y are digits) belong to Experiment X. After running each configuration, result can be evaluated by executing the `ExperimentX.R` file with the R statistical program.

Experiment 1

This experiment shows how the resources of a network can be allocated to different forwarding classes, and also demontrates the effect of these allocation on the quality of the service.

In configurations `Exp11–Exp15`, the traffic from hosts `H1–H4` are marked with different AFx1 classes at the edge routers. These forwarding classes are implemented by PPP queues using a WRR scheduler. The weights of the different forwarding classes are changed from the most polarized 7/6/5/4 to the least polarized 1/1/1/1. In the `Exp16` configuration, packets of the `H4` host are classified as EF, so they have higher priority at forwarding. The last configuration (`Exp17`), describing a Best Effort network, serves as a reference point for other measurements.

Results shows that packets that belong to different AFxy forwarding classes get different treatments, and the differences are larger when the variance of the weights is larger. It also can be observed that EF packets receives much better treatment than others: they have no packet loss, and have a very low delay.

Experiment 2

This experiment shows how the different drop precedencies of the AFxy forwarding classes can be implemented by RED queueing. In configurations Exp21–Exp24, hosts H1–H3 sends the normal voice and video traffic, while H4 sends only CBR traffic whose rate changes from 10kbps to 100kpbs in 10kpbs (so there is 10 measurement for each configuration). Packets from H1–H3 are marked with increasing drop priority (i.e. H1 traffic is forwarded as AF11, H2 as AF12, and H3 as AF13). The forwarding treatment of H4 traffic is changing in the different configurations: AF11 in Exp21, AF12 in Exp22, and AF13 in Exp23. The Exp24 configuration shows the behavior of a Best Effort network under the same traffic.

The plots shows that in the DiffServ network, there is no loss of AF11 traffic, and AF12 traffic loss is very low. The average queue length is lower in Exp23, because the red packets are dropped more likely.

Experiment 3

This experiment tests bandwidth utilization and flow isolation. Only hosts H1 and H2 generate traffic. H1 has an extra UDPBasicApp application, that generates a CBR traffic in addition to the voice and video streams. The rate of this extra traffic varies from 10kbps to 50kbps in 10kpbs steps in different measurements. Because the PPP links have only 300kbps datarate, the link between R2 and R3 is congested when the extra traffic is high enough.

With the first configuration (Exp31), both traffic policing and DiffServ queues are used. Traffic is metered at the egde routers by a token bucket meter. Packets whose rate is below 150kbps marked as AF11, the excess traffic is marked as AF12. RED queues in the network are configured so, that AF12 packets are dropped more aggressively than AF11 packets.

With the second configuration (Exp32), only traffic policing is used. Packets whose rate exceeding the allowed 150kbps limit are dropped at the boundary of the domain. There is no differentiated forwarding treatment of packets inside the domain. Each packet goes to the same queue in the PPP interfaces of routers. They are simple drop tail queues with 200 frames capacity.

The third configuration describes a Best Effort network. There is not traffic policing, and the PPP interfaces contain drop tail queues having buffer space for 200 frames.

As it is expected, the bandwith utilization is high in the first and third case, but lower in the second case. On the other hand, the Best Effort network does not provide the same flow isolation as the DiffServ network. In the third case the well-behaving H3 node suffers from the extra traffic generated by H1.

Experiment 5

This experiment compares a DiffServ network and a Best Effort network. In the DiffServ network, all voice traffic is marked with EF class at the edge routers. The video traffic originating from H1 and H3 are marked with AF11, while video traffic from H2 and H4 are marked with AF21. Routers contain DiffServQueue that implements the EF and AFxy forwarding classes with priority queueing, and weighted round-robin queueing. The weight of AF11 is a bit larger, than the weight of AF21, so hosts H1 and H3 gets better services. The queues contain a 100 frame buffer capacity for each forwarding class.

In the Best Effort network each PPP queue is a DropTailQueue. The queue capacities are 200 frames, to keep the buffer space equal to the DiffServ case.

In the DiffServ network, there is no loss of voice packets. Losses of video packets are smaller for `H1` and `H3` hosts, and higher for `H2` and `H4` than in a Best Effort network. This is expected, because DiffServ cannot increase the available resources, it only distributes them differently. It can also be observed, that the delay of voice packets are much smaller in the DiffServ network, so it can provide the necessary services for VoIP applications, while a Best Effort network can not.

Multiple domains example

This example describes a real-world scenario, and shows how the SLAs can be implemented with the components offered by INET.

Chapter 18

The MPLS Models

18.1 Overview

Blah blah blah

18.2 MPLS/RSVP/LDP Model - Implemented Standards

The implementation follows those RFCs below:

- RFC 2702: Requirements for Traffic Engineering Over MPLS
- RFC 2205: Resource ReSerVation Protocol
- RFC 3031: Multiprotocol Label Switching Architecture
- RFC 3036: LDP Specification
- RFC 3209: RSVP-TE Extension to RSVP for LSP tunnels
- RFC 2205: RSVP Version 1 - Functional Specification
- RFC 2209: RSVP Message processing Version 1

18.3 MPLS Operation

The following algorithm is carried out by the MPLS module:

```
Step 1: - Check which layer the packet is coming from
Alternative 1: From layer 3
    Step 1: Find and check the next hop of this packet
    Alternative 1: Next hop belongs to this MPLS cloud
        Step 1: Encapsulate the packet in an MPLS packet with
        IP_NATIVE_LABEL label
        Step 2: Send to the next hop
        Step 3: Return
```

```
    Alternative 2: Next hop does not belong to this MPLS cloud
        Step 1: Send the packet to the next hop
Alternative 2: From layer 2
    Step 1: Record the packet incoming interface
    Step 2: - Check if the packet is for this LSR
    Alternative 1: Yes
        Step 1: Check if the packet has label
        Alternative 1: Yes
            Step 1: Strip off all labels and send the packet to L3
            Step 2: Return
        Alternative 2: No
            Step 1: Send the packet to L3
            Step 2: Return
    Alternative 2: No
        Step 1: Continue to the next step
    Step 3: Check the packet type
    Alternative 1: The packet is native IP
        Step 1: Check the LSR type
        Alternative 1: The LSR is an Ingress Router
            Step 1: Look up LIB for outgoing label
            Alternative 1: Label cannot be found
                Step 1: Check if the label for this FEC is being requested
                Alternative 1: Yes
                    Step 1: Return
                Alternative 2: No
                    Step 1: Store the packet with FEC id
                    Step 2: Do request the signalling component
                    Step 3: Return
            Alternative 2: Label found
                Step 1: Carry out the label operation on the packet
                Step 2: Forward the packet to the outgoing interface found
                Step 3: Return
        Alternative 2: The LSR is not an Ingress Router
            Step 1: Print out the error
            Step 2: Delete the packet and return
    Alternative 2: The packet is MPLS
        Step 1: Check the LSR type
        Alternative 1: The LSR is an Egress Router
            Step 1: POP the top label
            Step 2: Forward the packet to the outgoing interface found
            Step 3: Return
        Alternative 2: The LSR is not an Egress Router
            Step 1: Look up LIB for outgoing label
            Alternative 1: Label cannot be found
                Step 1: Check if the label for this FEC is being requested
                Alternative 1: Yes
                    Step 1: Return
                Alternative 2: No
                    Step 1: Store the packet with FEC id
                    Step 2: Do request the signalling component
                    Step 3: Return
            Alternative 2: Label found
```

```
                Step 1: Carry out the label operation on the packet
                Step 2: Forward the packet to the outgoing interface found
                Step 3: Return
Step 2: Return
```

18.4 LDP Message Processing

The simulation follows message processing rules specified in RFC3036 (LDP Specification). A summary of the algorithm used in the RFC is presented below.

18.4.1 Label Request Message processing

An LSR may transmit a Request message under any of the conditions below:

- The LSR recognizes a new FEC via the forwarding tale, and the next hop is its LDP peer. The LIB of this LSR does not have a mapping from the next hop for the given FEC.
- Network topology changes, the next hop to the FEC is no longer valid and new mapping is not available.
- The LSR receives a Label Request for a FEC from an upstream LDP and it does not have label binding information for this FEC. The FEC next hop is an LDP peer.

Upon receiving a Label Request message, the following procedures will be performed:

```
Step 1: Extract the FEC from the message and locate the incoming interface
        of the message.
Step 2: Check whether the FEC is an outstanding FEC.
    Alternative 1: This FEC is outstanding
        Step 1: Return
    Alternative 2: This FEC is not outstanding
        Step 1: Continue
Step 3: Check if there is an exact match of the FEC in the routing table.
    Alternative 1: There is an exact match
        Step 1: Continue
    Alternative 2: There is no match
        Step 1: Construct a Notification message of No route and
                send this message back to the sender.
Step 4: Make query to local LIB to find out the corresponding label.
    Alternative 1: The label found
        Step 1: Construct a Label Mapping message and send over
                the incoming interface.
    Alternative 2: The label cannot be found for this FEC
        Step 1: Construct a new Label Request message and send
                the message out using L3 routing.
        Step 2: Construct a Notification message indicating that the
                label cannot be found.
```

18.4.2 Label Mapping Message processing

Upon receiving a Label Mapping message, the following procedures will be performed:

```
Step 1: Extract the FEC and the label from the message.
Step 2: Check whether this is an outstanding FEC
    Alternative 1: This FEC is outstanding
        Step 1: Continue
    Alternative 2: This FEC is not outstanding
        Step 1: Send back the server an Notification of Error message.
Step 3: Install the new label to the local LIB using the extracted label,
        FEC and the message incoming interface.
```

18.5 LIB Table File Format

The format of a LIB table file is:

The beginning of the file should begin with comments. Lines that begin with # are treated as comments. An empty line is required after the comments. The "LIB TABLE" syntax must come next with an empty line. The column headers follow. This header must be strictly "In-lbl In-intf Out-lbl Out-intf". Column values are after that with space or tab for field separation. The following is a sample of lib table file.

```
#lib table for MPLS network simulation test
#lib1.table for LSR1 - this is an edge router
#no incoming label for traffic from in-intf 0 &1 - LSR1 is ingress router for those tra
#no outgoing label for traffic from in_intf 2 &3 - LSR 1 is egress router for those tra

LIB TABLE:

In-lbl  In-intf         Out-lbl     Out-intf
1       193.233.7.90    1           193.231.7.21
2       193.243.2.1     0           193.243.2.3
```

18.6 The CSPF Algorithm

CSPF stands for Constraint Shortest Path First. This constraint-based routing is executed on-line by Ingress Router. The CSPF calculates an optimum explicit route (ER), based on specific constraints. CSPF relies on a Traffic Engineering Database (TED) to do those calculations. The resulting route is then used by RSVP-TE.

The CSPF in particular and any constraint based routing process requires following inputs:

- Attributes of the traffic trunks, e.g., bandwidth, link affinities
- Attributes of the links of the network, e.g. bandwidth, delay
- Attributes of the LSRs, e.g. types of signaling protocols supported
- Other topology state information.

There has been no standard for CSPF so far. The implementation of CSPF in the simulation is based on the concept of "induced graph" introduced in RFC 2702. An induced graph is analogous to a virtual topology in an overlay model. It is logically mapped onto the physical network through the selection of LSPs for traffic trunks. CSPF is similar to a normal SPF, except during link examination, it rejects links without capacity or links that do not match color constraints or configured policy. The CSPF algorithm used in the simulation has two phases. In the first phase, all the links that do not satisfy the constraints of bandwidth are excluded from the network topology. The link affinity is also examined in this phase. In the second phase, Dijkstra algorithm is performed.

Dijkstra Algorithm:

```
Dijkstra(G, w, s):
   Initialize-single-source(G,s);
   S = empty set;
   Q = V[G];
   While Q is not empty {
       u = Extract-Min(Q);
       S = S union {u};
       for each vertex v in Adj[u] {
           relax(u, v, w);
       }
   }
```

In which:

- G: the graph, represented in some way (e.g. adjacency list)
- w: the distance (weight) for each edge (u,v)
- s (small s): the starting vertex (source)
- S (big S): a set of vertices whose final shortest path from s have already been determined
- Q: set of remaining vertices, Q union S = V

18.7 The traffic.xml file

The traffic.xml file is read by the RSVP-TE module (RSVP). The file must be in the same folder as the executable network simulation file.

The XML elements used in the "traffic.xml" file:

- `<Traffic></Traffic>` is the root element. It may contain one or more `<Conn>` elements.
- `<Conn></Conn>` specifies an RSVP session. It may contain the following elements:
  - `<src></src>` specifies sender IP address
  - `<dest></dest>` specifies receiver IP address
  - `<setupPri></setupPri>` specifies LSP setup priority
  - `<holdingPri></holdingPri>` specifies LSP holding priority
  - `<bandwidth></bandwidth>` specifies the requested BW.

- `<delay></delay>` specifies the requested delay.
- `<route></route>` specifies the explicit route. This is a comma-separated list of IP-address, hop-type pairs (also separated by comma). A hop type has a value of 1 if the hop is a loose hop and 0 otherwise.

The following presents an example file:

```
<?xml version="1.0"?>
<!-- Example of traffic control file -->
<traffic>
   <conn>
       <src>10.0.0.1</src>
       <dest>10.0.1.2</dest>
       <setupPri>7</setupPri>
       <holdingPri>7</holdingPri>
       <bandwidth>400</bandwidth>
       <delay>5</delay>
   </conn>
   <conn>
       <src>11.0.0.1</src>
       <dest>11.0.1.2</dest>
       <setupPri>7</setupPri>
       <holdingPri>7</holdingPri>
       <bandwidth>100</bandwidth>
       <delay>5</delay>
   </conn>
</traffic>
```

An example of using RSVP-TE as signaling protocol can be found in ExplicitRouting folder distributed with the simulation. In this example, a network similar to the network in LDP-MPLS example is setup. Instead of using LDP, "signaling" parameter is set to 2 (value of RSVP-TE handler). The following xml file is used for traffic control. Note the explicit routes specified in the second connection. It indicates that the route is a strict one since the values of every hop types are 0. The route defined is 10.0.0.1 -> 1.0.0.1 -> 10.0.0.3 -> 1.0.0.4 -> 10.0.0.5 -> 10.0.1.2.

```
<?xml version="1.0"?>
<!-- Example of traffic control file -->
<traffic>
    <conn>
        <src>10.0.0.1</src>
        <dest>10.0.1.2</dest>
        <setupPri>7</setupPri>
        <holdingPri>7</holdingPri>
        <bandwidth>0</bandwidth>
        <delay>0</delay>
        <ER>false</ER>
    </conn>
    <conn>
        <src>11.0.0.1</src>
        <dest>11.0.1.2</dest>
```

```
        <setupPri>7</setupPri>
        <holdingPri>7</holdingPri>
        <bandwidth>0</bandwidth>
        <delay>0</delay>
        <ER>true</ER>
        <route>1.0.0.1,0,1.0.0.3,0,1.0.0.4,0,1.0.0.5,0,10.0.1.2,0</route>
    </conn>
</traffic>
```

Chapter 19

Applications

19.1 Overview

This chapter describes application models and traffic generators.
Blah blah blah

Chapter 20

History

20.1 IPSuite to INET Framework (2000-2006)

The predecessor of the INET framework 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 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, Andras Varga took over the development in July 2003. Through a series of snapshot releases in 2003-2004, modules got completely reorganized, documented, and many of them rewritten from scratch. The MPLS models (including RSVP-TE, LDP, etc) also got refactored and merged into the codebase.

During 2004, Andras added a new, modular and extensible TCP implementation, application models, Ethernet implementation and an all-in-one IP model to replace the earlier, modularized one.

The package was renamed INET Framework in October 2004.

Support for wireless and mobile networks got added during summer 2005 by using code from the Mobility Framework.

The MPLS models (including LDP and RSVP-TE) got 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 got merged into the INET CVS in fall 2005, and got 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 got 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 fall 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 got 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 was contributed by Thomas Reschka.

Several other people have contributed to the INET Framework by providing feedback, reporting bugs, suggesting features and contributing patches; I'd like to acknowledge their help here as well.

References

[CBD02] Tracy Camp, Jeff Boleng, and Vanessa Davies. A survey of mobility models for ad hoc network research. *WIRELESS COMMUNICATIONS & MOBILE COMPUTING (WCMC): SPECIAL ISSUE ON MOBILE AD HOC NETWORKING: RESEARCH, TRENDS AND APPLICATIONS*, 2:483–502, 2002.

[Chi98] Ching-Chuan Chiang. Wireless network multicasting, 1998.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance, 1993.

[Pea99] Charles E. Perkins and et al. Optimized smooth handoffs in mobile ip. In *IN PROCEEDINGS OF ISCC*, pages 340–346, 1999.

[Ram10] Sanjay Ramroop. Performance evaluation of diffserv networks using the ns-3 simulator, 2010.

Index