# Abstract:

Recognizing the need for a network simulator, that is capable of handling large network simulations running over large simulation time spans, requiring a multitude of resources to achieve the desired state of accuracy, due to the scale and heterogeneity of the topologies under consideration. And to allow for better assessment of real life threats to network architecture from various events such as natural disasters, and man caused events. A network simulator based upon the DEVS formalism was proposed to fill the need for such a tool. The formalism is based on sound theoretical grounds, allowing for an abstract design of models that would be independent from the implementation platform and running conditions. It will also enable the simulator to maximize resource use by means of distributing the load of the simulation.

CD++ was chosen as the tool for implementing the library models, providing the ground bases for the simulator. Furthermore, the tool follows the theoretical bases of the DEVS formalism, allowing the models to be run in a parallel distributed environment, which would result in a decreased simulation time as a whole and better resource management schemes.

The models are built in a modular fashion to maintain a high degree of flexibility and customizability for future development of the tool. Further more, the models are built to be as generic as possible, so as to leave a space for development of other network devices and protocols using the existing models as templates and guidelines.

# 1.0 Introduction

The fourth year project report submitted by Mohaemd Abd El-Salam, Khalil Yonis and, Abdul-Rahman Elsahfei, titled *"building a library for parallel simulation of networking protocols"* aims at shedding the light on the details of the design, implementation, and testing of the DEVS models comprising the library. The library facilitates the simulation of complex network architectures, built on the TCP/IP protocol stack. This document, in addition to being required for the successful competition of our fourth year project, is also necessary, for projects aiming to continue working on extending the model library in particular, and building a network simulator tool based on the DEVS formalism in general.

Rapid diffusion of internetworking technology brings two major sources of stress to the underlying protocol mechanisms and associated design methods: scale, and heterogeneity. Scale, affects both the correctness and the performance of a network in general. On the other hand, Heterogeneity of applications translates into a large number of interacting protocols, each with a certain requirements and traffic pattern [1].

The dynamic behaviour of networking protocols in packet-switched data networks must be examined to determine if current protocol design and engineering practices are critically adequate, to produce robust and evolvable network technology in the future.

To be able to handle large-scale simulations, a network simulator based on the DEVS formalism was proposed. The simulator should be capable of simulating user-defined topologies to assess network functionality. The simulator should be built upon a modular library of models defining the behaviour of well-known protocol stacks (such as TCP/IP), and common widely used inter-networking devices. Also, the library modular design should allow for the addition of new models easily and the models themselves should be flexible enough to allow future enhancements.

Although various network simulators are readily available (both academic and commercial such as OPNET, OMNet++, and NS-2), it was felt that a new simulation library based on the DEVS formalism, would add the ability to interface models simulating non-network entities, that affect network operation to a network topology, to assess their effect on network operation, thus coming up with more realistic results from the simulation.

The rest of the report will go through a background chapter explaining the bases on which the library models were designed, with a brief explanation of the tool used. The background, shall then explain what the main functionality of each model, and the packet format used as traffic in the network. After this, the design and implementation chapter will show the DEVS model specification for each library device and show the implementation of the each of the devices. To prove the functionality of the library, the testing chapter will explain some of he tests ran to verify the behaviour of the models, and the integration tests performed.

Finally, the report will provide a recommendations chapter outlining steps that should be taken for future extensions. The recommendations aim at getting the project closer to becoming a complete network simulator based on the DEVS formalism.

# 2.0 Background

The background section will present some details concerning the tool used in the project. The chapter also define the components comprising the library, giving an overview of the functionality of each model. For simplicity, each set of models was grouped together to assemble a specific device. Never the less, the models in general were built to be generic, meaning models from one device could be used in another according to the designer's needs. And the last section of this chapter will discuss the choices made in creating the packet that will represent out network traffic.

The library consists of two major units, data generators and inter-networking devices. Data generators are modeled with the host model. The model is based on emulating the TCP/IP protocol stack. Inter-networking devices models are a router and a hub, which gives the library the initial depth it needed to simulate fairly complex topologies.

Before the project went under way, it was felt that a survey about field of parallel simulation was necessary, to acquaint our selves with the latest developments in the field. The research documentations are provided in appendix "A". We also studied currently available network modeling and simulation tools, focusing on OPNET, OMNet++, and NS-2. The research in particular was very helpful in choosing what devices to include in this project, so as to serve as a starting point for an ongoing project. The surveys compiled can also be found in appendix "A".

The remaining of this chapter will introduce the DEVS formalism, and the main features of the tool that our work was based on (CD++). CD++ is a modular based simulation tools, built on the DEVS formalism. The tool allows for building event driven models to simulate complicated systems. Finally the chapter intends to explain the functionality of each model in the library, and the traffic format.

## *2.1 The DEVS formalism and the CD++ toolKit*

DEVS (**D**iscrete-**Ev**ent system **S**pecifications) was developed as a theoretical approach, which allows the definition of hierarchical models that can be easily integrated and reused [2]. Any system modeled with DEVS is described as a composite of sub-models, each being an Atomic or a coupled model.

DEVS Atomic models are formally defined as follows:

$$\mathbf{M=<I, X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D>}$$

Where:

**I**: set of model interfaces

**X**: the input events set.

**S**: the state set.

**Y**: the output event set.

$\mathbf{\delta_{int}}$: the internal transition function.

$\mathbf{\delta_{ext}}$: the external transition function.

$\mathbf{\lambda}$: the output function.

**D**: the duration function.

Each atomic model is provided with a set of unidirectional ports (input and output) to interact and communicate with other models. The input event set and the output event set are made of all possible events that might occur on the input or output ports respectively. The external function is invoked when an event occur on any of the model input ports. In the external function (described later on) the event gets processed and the model executes by changing variables and setting states if needed as a result of the event.

The model stays in its current state for a period defined by the duration function. When the duration function time expires, the output function is invoked. The output function sends events from the output event set through specific output ports, defined in the model's set of outputs, according to the models current state.

After the output function execution, the internal transition function will be invoked to determine the new state of the model. The duration function is invoked before every execution of the external and the internal function, since every state must be associated with a unique timing value.

A coupled DEVS model is a set of interconnected basic models (atomic or coupled). The coupled model is defined formally by:

$$\mathbf{CM <I, X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}>}$$

Where:

**I**: the models interface.

**X** the input event set;

**Y** the output event set;

**D** the index of the components of the coupled model; and $\forall$ i $\in$ D,

$\mathbf{M_i}$ is the a basic DEVS (an atomic or coupled model);

$\mathbf{I_i}$ is the set of influences of model i (that is the model that can be influenced by model i); and $\forall$ j $\in$ $I_i$,

$\mathbf{Z_{ij}}$ is the i to j translation function.

DEVS coupled models are defined by a set of inter-connected atomic or coupled models. The influencees of a model determine where to send the outputs. The translation function is in charge of converting one model's outputs into another model’s inputs. To do so, an index of influencees ($I_i$) is kept to determine which outputs of models $M_i$ are connected to inputs of model Mj, where j is an element of $I_i$.

CD++ is a toolkit for modeling and simulation based on the DEVS formalism. The tool is built as a set of independent software pieces running on different platforms [3]. The toolkit depends on the concept of separating the modeling process from the simulation. Atomic models are built in C++, and coupled models are defined using a specification language. The language provides a textual representation independent from any tool and development environment [3].

Atomic models are created using C++ classes, derived from the class *Atomic*. The new class representing a model must overwrite four functions inherited from class *Atomic*, to define the behavior of the model. Coupled models, on the other hand are a combination of models either Atomic or coupled, with the addition of the Top-level port connections added.

For each atomic model two sets of ports are defined; a set of input ports to receive incoming events, and a set of output ports to send outgoing event. Input and output ports are defined first in the model header file, as follows

```
private:
    const Port &in;          // defining input port “in”
    Port &out;               // defining output port “out”
```

**Figure 1: Atomic model’s port definition**

After a port has been defined in the header file, it is registered with the tool by calling on the add port method as shown below.

```
modelName :: modelName(const string &name):Atomic(name)  // Constructor
,in( addInputPort ("in") )              // register Input port "in"
,out( addOutputPort ("out') )           // register Output port "out"
{….}
```

**Figure 2: Atomic model port registration**

As mentioned earlier, each model must overwrite four functions, which represents the DEVS specification in the CD++ toolkit. The functions are:

- **initFunction**: this function starts executing with the start of the simulation. In this function the initial values for the model variables are set, and the state is normally set to passive.
- **externalFunction**: this function is invoked every time an external event is detected at any of the model's input ports. Normally each port will have an associated action to perform.
- **outputFunction**: this function is invoked after the duration function has expired. This is the only place where the model should interact with other models by means of outputting messages, to conform to the DEVS specifications.
- **internalFunction**: this function is invoked upon expiry of the duration function and after the outputFunction has finished execution.

The four previously mentioned functions could make use of a set of methods defined in the tool, to manipulate simulation time and state, including:

- **holdIn ( state, time)**: instructs the model to hold in the specified **state** for certain amount of **time**. The expiration of the **time** invokes the outputFunction and the internalFunction.
- **passivate()**: sets the atomics model state to the passive state. Equivalent to calling holdIn(passive, infinity).
- **sendOutput( time, port, value)**: this function is called from the model's outputFunction to send out events. In this function the output **port** that will be used and the **value** for the event, and the event time are specified. The **value** must be a double.
- **state()**: returns the current state of the atomic model.

After building a new model, we must incorporate it with the rest of the tool. In order to do that we must change the file “register.cpp”, by adding a call to the

singleModelAdm::regisrteAtomic method from within the MainSimulator class registerNewAtomics method. This is shown in figure 3.

```
#include "modelHeaderfile.h"

void MainSimulator::registerNewAtomics()
{
SingleModelAdm::Instance().registerAtomic(NewAtomicFunction<modelName>()
, "modelName" ) ;
}
```

**Figure 3: reistering a new atomic model**

As we need the new model to be part of the tool, the compiler must be instructed to compile the new model's code files (header and source code files) as an integral part of the simulator. The compilation of the simulator is done by means of the "make" command, which searches for a "makefile". The makefile will instruct the compiler which files to use to create the simulator executable file. To make the tool compile the new model files, the models must be added to the "makefile". Changes to the makefile are shown below.

```
DEFINES_C=


# If we are compiling for Unix
INCLUDES_CPP=-I/usr/include
# or if we are compiling for Windows   95
#INCLUDES_CPP=


INCLUDES_C=
CFLAGS=
DEBUGFLAGS =
LDFLAGS  +=-L. -g

EXAMPLESOBJS=MyModel.o queue.o main.o generat.o cpu.o transduc.o trafico.o distri.o com.o
linpack.o debug.o register.o
LIBNAME=simu
LIBS=-lsimu
ALLOBJS =${EXAMPLESOBJS } ${SIMOBJS}
INIOBJS=initest.o ini.o
ALLSRCS=${ALLOBJS:.o=.cpp} gram.y
.
.
.
.
.

############################
# Without Optimization
MyModel.o: MyModel.cpp
        ${CPP} -c ${INCLUDES_CPP} ${DEFINES_CPP} ${DEBUGFLAGS} ${CPPFLAGS} $<

generat .o: generat .cpp
        ${CPP} -c ${INCLUDES_CPP} ${DEFINES_CPP} ${DEBUGFLAGS } ${CPPFLAGS} $<

queue .o: queue .cpp
        ${CPP} -c ${INCLUDES_CPP} ${DEFINES_CPP} ${DEBUGFLAGS } ${CPPFLAGS} $<

toCDPP.o: toCDPP.cpp
        ${CPP} -c ${INCLUDES_CPP} ${DEFINES_CPP} ${DEBUGFLAGS } ${CPPFLAGS} $<

mainsimu.o: mainsimu.cpp
        ${CPP} -c ${INCLUDES_CPP} ${DEFINES_CPP} ${DEBUGFLAGS } ${CPPFLAGS} $<

# Uncomment these lines only for Windows
#macroexp .o: macroexp.cpp
#       ${CPP} -c ${INCLUDES_CPP} ${DEFINES_CPP} ${DEBUGFLAGS } ${CPPFLAGS} $<
#
#flatcoup .o: flatcoup .cpp
#       ${CPP} -c ${INCLUDES_CPP} ${DEFINES_CPP} ${DEBUGFLAGS } ${CPPFLAGS} $<
#############################

..
.

.
.
.
.
```

**Figure 4: Makefile changes**

Once the model is integrated with the simulator, a simulation can be executed. The simulator receives a set of inputs to execute (a model file and an optional external event file), an output and a log file (used to show the simulation outputs, and in case of errors the log file can be viewed to determine where the fault occurred).

The simulator is activated by

```
./simu –mMyModel.ma –eMyModel.ev –oMyModel.out –lMyModel.log
```

**Figure 5: Simulator activation command**

After compiling the models, the new models can be instantiated and used within the model file (*.ma). In the model file coupled models are created by linking atomic models together. A simple example of model file is shown below.

```
[top]
components : router_out@RouterOutput
out  : out
in   : from_RPU interfaceNum
link : out@router_out out
link : interfaceNum interfaceNum@router_out
link : from_RPU from_RPU@router_out

[router_out]
preparation : 00:00:00:050
```

**Figure 6: the RouterOut model file**

The model file is made up of at lest one component called the **top**. Components in the model file are defined between two square brackets and follow a specific format in there definition. After the name of the component is specified, a list of sub-components is defined after the key word **components:** every sub-component is either an instance of an atomic model or another component. The format for defining sub-components is:

**Instatnce_name@atomic_model_name** for instances of atomic models,

or

**component_name** for components that are other coupled model in the model file.

Once listing the models components is done, the list of input and output ports for the model is defined, each after its keywords **in:** and **out:** respectively. Once the models ports are defined, the linking of models begins by using the keyword link: followed by the port that will output the event then the port that will receive the event as shown in the example below.

**Link : source_port@instance_name destination_port@other_instance_name**

If any of the ports used belongs directly to the model and not to one of its components, the identifier part (@instatnce_name) of the syntax is dropped.

Link : source_port @instance_name **destination_port**

Link : **source_port** destination_port @instance_name

The event file is used to input events to the simulation at specific times. Values coming from the event file are used to excite the system, to observe its behaviour. Event files are simply a list of time variables that defines the event time, followed by the port that the event must arrive at, then the value of the event. Something to note here is that event file's can only support double values for their events. The format for the event file is shown in the figure below.

| time(h:m:s:ms) | port | event |
|---|---|---|
| 00:00:06:002 | in | 2.3 |
| 00:00:06:003 | in | 2.4 |
| 00:00:07:000 | ready | 3 |

**Figure 7: event file example**

After this brief introduction about the tool and the formalism used in the course of this project, the remaining of the background chapter describes the behaviour of the various models of each of the library devices.

## *2.2 Library models*

Models comprising the library where chosen very carefully. The criteria was to choose models, which will enable the creation of fairly complex network topologies, and in the same time to function as a starting point for other projects, aiming at incrementing the functionality of the library, to achieve the goal of creating a network simulator based on the DEVS formalism. It must be noted, that flexibility of the models was of great importance, the models where designed to be as generic as possible, to allow each model to be reused in many devices if needed.

As mentioned earlier in this chapter, the library consists of a Host as a representation of a data generator and a couple of inter-networking devices (the router and the hub). The remaining of this section will describe the function of each one of these models.

### 2.2.1 Host

The Host emulates the behaviour of the TCP/IP protocol stack. This specific stack was chosen for its wide use, and abundance of information related to it. The Stack is divided into five layers, outlined in the following figure:

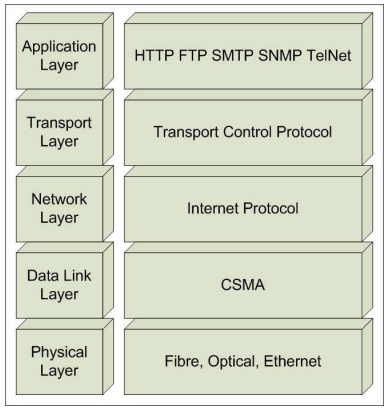


**Figure 8: TCP/IP protocol stack**

The application layer is the top layer of the network protocol stack. It is concerned with the semantics of work, and how to represent that data [4].

The transport Layer transport data streams from a source application to a destination application reliably, and with integrity. The layer is capable of handling multiple connections and multiple applications. The Transport layer isolates lower layers from application programmers.

The transport layer could implement one of two protocols: Transport Control Protocol (TCP) or the user datagram protocol (UDP). In this project the emphasis was on TCP, which is a connection oriented protocol that provides a reliable data transmission via end-to-end error detection and correction. TCP Guarantees that the data is transferred across a network accurately and in the proper order. The protocol retransmits any data not received by the destination node, it also guarantees against data duplication between sending and receiving nodes. Finally TCP supports Telnet, FTP, SMTP, and POP [5]. TCP is discussed in RFC# 793[6]

The network protocol is the heart of the TCP/IP protocol stack. The protocol “IPV4” was chosen as the network protocol, for its heavy use in both the commercial and industrial sectors, making it the most widely used network protocol the Internet is made of. Another reason for this choice is the development of “IPV6” which having the same architecture of “IPV4” adds more quality of service parameters and more addressing space. By having a model of “IPV4” this allows the simulation of current packet switched networks, with the ability to create an “IPV6”, later on, to complement the protocol library.

“IPV4” is a classless addressing protocol (meaning The protocol header carries its full addressing parameters such as source and destination addresses), the protocol also holds Quality of service (QOS) parameters for routing purposes such as time to live and identification field to allow for fragmentation at routers. In addition, it includes error

control parameters (Header checksum field). Finally, the protocol provides full-duplex communications of the network.

The Data Link layer is associated with logical interface between an end system and a network. It is responsible for providing the means to activate, maintain, and deactivate the link. It also provides services to the higher layers of the TCP/IP protocol architecture such as error detection and control. The structure of the data link control is divided into two sub-layers; the Logical Link Control and the Medium Access Control outlined in the following figure. [7]

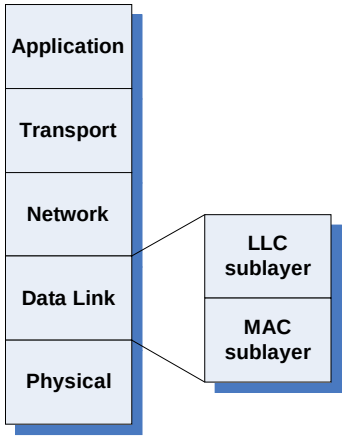


**Figure 9: Data Link sub layers**

The Logical Link Control provides an interface to higher layers and performs error detection and control. On the other hand, the Medium Access Control sub-layer provides controlling access to the transmission medium in order to give an orderly and efficient use of that capacity.

Whenever higher-level data is sent to the data-link layer, the Logical link control creates a Protocol Data Unit (PDU) with control information appended to the data as a header. The LLC then keeps track of the PDU's that have been successfully received and retransmits unsuccessful frames [7:437]. The PDU format is shown in figure 8.

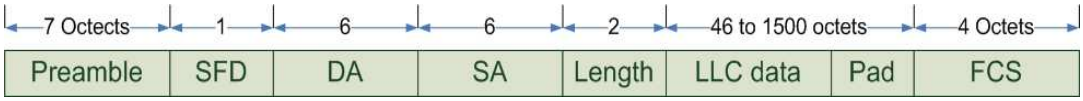


Figure 10: LLC PUD

The fields of the PDU are:

1. **I/G:** Individual/Group
2. **DSAP**: destination service access points
3. **C/R**: Command/Response
4. **SSAP:** Source service access points
5. **LLC control:** specifies the type of frame
6. **Information:** can either be control information or the Packet received from the internet layer.
7. **FCS:** Frame check Sequence for CRC error detection

DSAP and SSAP are addresses given to LLC users. I/G, C/R and LLC control are services that are mainly based on the High-level Data Link Control standards [7:437]. Most of these services are not discussed in this section because they are not a major concern, for this project. On the other hand, HDLC flow control mechanisms were already modeled in another protocol layer.

Error detection in the LLC was our main interest in-order to discard any frames that are in error. One of the most common error detecting techniques used in data link protocols is the Cyclic Redundancy Check [7:202]. The CRC can be described as follows:

“Given a k-bit block of bits, or message, the transmitter generates an n-bit sequence, known as a frame check sequence (FCS), so that the resulting frame, consisting of k + n bits, is exactly divisible by some predetermined number” [7:202]. Hence, an error can be detected in an incoming frame by dividing the frame by the predetermined number and examine if the remainder is greater than zero. These predetermined numbers, P(X) are usually expressed as polynomials with binary coefficients that correspond to the

bits in a binary number. One version of P(X) that is usually used in wide area networks is the CRC-16 [7:204]. A mathematical presentation of the CRC-16 process can be described as:

$$\text{CRC-16: } P(X) = X^{16} + X^{15} + X^2 + 1 = (1100000000000101)_b = (98309)_d$$

Let, M = original message or data

$\overline{M}$ = the message or data received

$\therefore$ FCS = M / P(X)

$\therefore$ Discard frame $\begin{cases} \text{Yes, if } (\overline{M} - \text{FCS}) \bmod P(X) = 0 \\ \text{No, otherwise} \end{cases}$

The Medium Access Control (MAC) sub layer uses the carrier sense multiple access with collision detection (CSMA/CD) control technique, which is the basis for the IEEE 802.3 standard [7:470]. Based on this technique, if a device wishes to transmit data, it first senses the carrier to find out if another device is transmitting data over the link. If the medium is busy, the device must wait, otherwise it may transmit data. After transmission of data, the device senses the carrier again if there has been a collision just in case if another device was sending data at the same time. If a collision has been detected, the device sends out a 32-bit jamming signal into the transmission link that informs all connected devices that a collision has occurred [8].

As a result of the jamming signal, all devices that have sent data over the transmission link during the collision would resend the data again but after a random delay. The random delay ensures that the retransmission of data from the connected devices does not occur at the same time to avoid simultaneous collisions [8]. A flow chart describing the CSMA/CD algorithm is in the following figure.

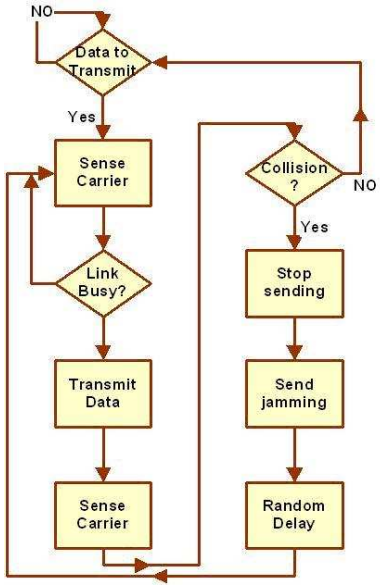


**Figure 11: CSMA**

Besides the CSMA/CD operation, receives a PDU from the LLC and appends a header to create a MAC frame. The MAC frame shown in figure 12 has the following fields:

- preamble: pattern of alternating ones and zeros and 1's used by receiver for synchronization
- Start frame delimiter (SFD): to locate first bit of rest of frame
- Pad: octets added to ensure that the frame is long enough for proper operation
- Source Address (SA): the station that sent the frame
- Destination Address (DA): the physical address of the destination
- Length/Type: length of the LLC PDU
- LLC: the PDU sent from the LLC sub-layer

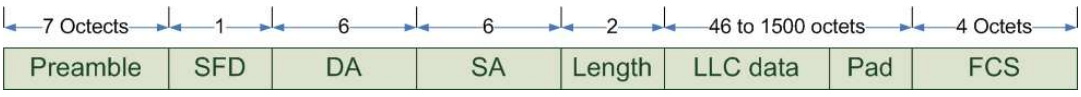


**Figure 12: MAC frame**

The FCS in the MAC frame is provided by the LLC sub-layer, which was discussed earlier. Similar to the LLC, the other minor functions used in the MAC frame

are based on bit-operations and it is difficult to model in CD++, since message passing in CD++ are of double variables. Details of this problem is discussed in the problems encountered section

The physical layer is the lowest layer of both the ISO/OSI and TCP/IP protocol stacks. Consists of the cables, connectors and associated hardware such as driver chips to implement a network such as Ethernet or Token Ring [9]. For the purpose of this project, the physical layer will be limited to three of the widely used implementations Fibre optics, T1, and Ethernet.

T1 is known to be "a digital transmission link with capacity of 1.544 Mbps. T1 uses two pairs of twisted pairs of normal twisted wires, the same found in most residences. T1 normally handles 24 voice conversations, each one digitized at 64 Kbps. But, with more advanced digital voice encoding techniques, it can handle more voice channels. T1 is a standard for digital transmission in the United States. T1 lines are used to connect networks across remote distances. Hubs and routers are used to connect LANs over T1 networks."[10]

Fibre optics on the other hand, is "the technology in which communication signals in the form of modulated light beams are transmitted over a glass fibre transmission medium. Fibre optic technology offers high bandwidth, small space needs and protection from electromagnetic interference, eavesdropping and radioactivity"[11].

The last physical layer implementation we are interested in, is the Ethernet Link, which is "a very common method of networking computers in a LAN. Ethernet will handle about 10 Mbits-per-second. And can be used with almost any kind of computer"[12].

### 2.2.2 Inter-networking devices

To be able to create and simulate network topologies we needed to include inter-networking devices to the library. The devices we decided to add are a router and a hub.

The router is the device that determines the next network point to which a data packet should be forwarded. Routers route information based on traffic's layer-three information (IP address). Routers maintain a table of the available routes and use this information to determine the best route for a given data packet [13]. The router extracts the packets destination IP address and compares it to the entries in its routing table, which contains the needed information for routing packets.

An Internet routing protocol enables exchanging information about reachability of destinations in the network. To dynamically update the routing information, special routing protocols are used. One of the first routing protocols used in (DARPA internet) was the Routing Information Protocol (RIP) [13].

RIP is a n interior routing protocol designed to work with IP-based moderate sized networks using a reasonably homogenous technology [14]. RIP uses the distance vector algorithm to find the best route with the smallest metric size for each destination. There for, keeping a table with an entry for every possible destination is necessary.

In order to gather the necessary information about the network topology, routers send two main messages to its neighbouring nodes; Request command to ask for routing information and to make sure that they are still functioning, and Response commands to respond on received requests from neighbours.

According to RIP, the request command is sent every 30 sec. to ensure that neighbour nods are still connected, and to gather routing information. If a neighbour node did not respond within 180 sec. the router will consider this nod to be disconnected.

As for the hub, it is a simple network device that joins multiple clients by means of a single link to the rest of the LAN. A hub has several ports to which clients are connected directly, and one or more ports that can be used to connect the hub to the backbone or to other active network components. The hub's operates as a multi-port repeater; signals received on any port are immediately retransmitted to all other ports of the hub. Hubs function at the physical layer of the reference model [15].

## 2.3 Traffic Format

The headers for the Internet Protocol are based on RFC # 791[6]. They contain the full addressing information (source and destination IP) as well as other Quality of Service parameters such as Time to live (TTL), identification field, and finally a checksum. The choice of these parameters was due to the fact that CD++ can only handle primitive types for the time being. This choice of parameters will enable us to create simple Service level Agreements (SLA) for a more realistic simulation of the Core network. However, a more complete version of the IP header is provided in Appendix "E". This version offers bit level manipulation of the header allowing the model to provide the full functionality of the protocol.

The traffic packets are made of four values; either the option or the update field followed by the source address, the destination address, and the TCP field. The options in each field chosen from the IPV4 packet format are presented in the following figure.

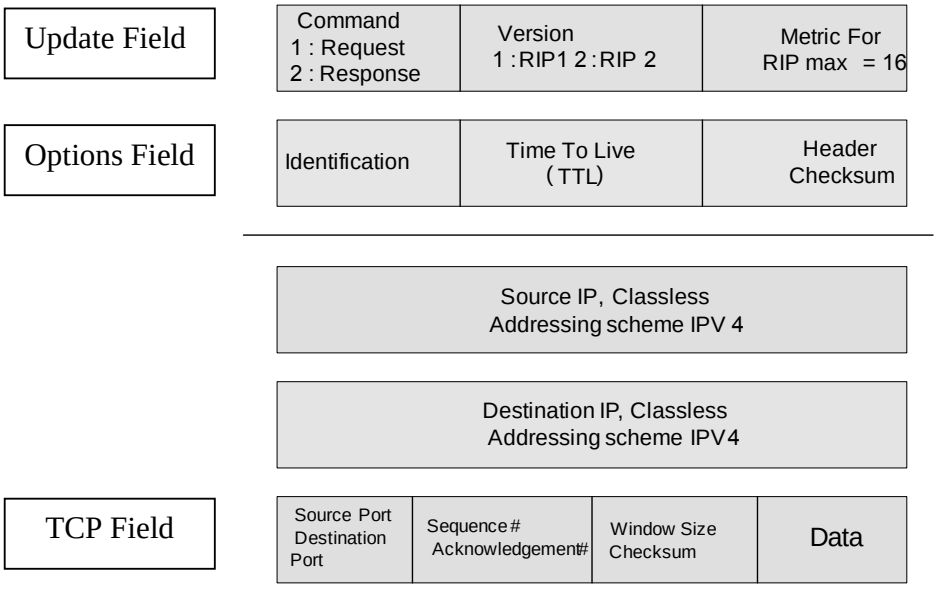


**Figure 13 IP headers Format**

Where:

- Command; Defines the update type either request or response, more on that.
- Version; Used for the RIP protocol, to identify protocol version (1 for RIP 1 and 2 for RIP 2).
- Metric; specifies the cost (number of hop) for getting to the specified destination (maximum of 16).
- Time to live; represent the maximum number of hobs the packet is allowed to take before it gets discarded.
- Identification; A field to identify packets belonging to the same transfer.
- Header checksum; a mathematical calculation to assure packet integrity.
- Source IP address; the IP address of the packet's source host.
- Destination IP address; the IP address of the destination of the packet.
- Source & destination ports; value identifying which application sent the data. And where it should be received
- Sequence number: The sequence number of the first data octet in this segment (except when SYN is present). If SYN is present the sequence number is the initial sequence number (ISN) and the first data octet is ISN+1 [16].
- Acknowledgment number; value of the next sequence number the sender of the segment is expecting to receive. Once a connection is established this is always sent [16].
- Window size; The number of data octets beginning with the one indicated in the acknowledgment field which the sender of this segment is willing to accept[16].
- Checksum; addition of the values in the packet
- Data; data portion to send.

# 3.0 Design and implementation

This chapter will show each model's DEVS specification and explain the behaviour and functionality of each of the models.

The three devices explained are the Host, the Router, and the Hub.

## *3.1 Host*

The host is comprised of five models. The models are built to be reusable, in any other device as need arises. The five models represent the host's application layer, the transport layer, network layer, the data link layer, and physical layer.

### 3.1.1 The Application layer

The front end of the host model is the application layer according to the TCP/IP protocol stack, illustrated in figure 8. It is designed as a simple atomic model as follows:

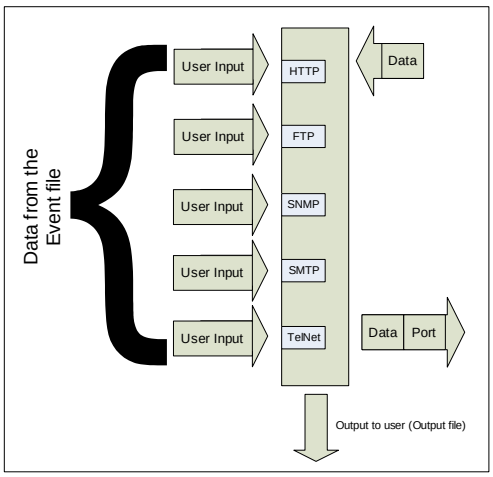


**Figure 14: Application logical Design**

The layer manipulates the data received from the user, in a way to identify application type sending the data. This step is done to facilitate creating a connection manager. The data specifications is shown here

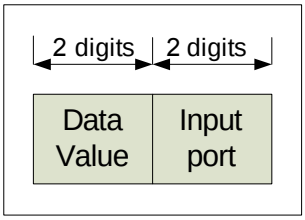


**Figure 15: Application Layer otput data format**

The layer formal specification is as follows:

$$M=<I, X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D>$$

Where:

I(interface):

HTTP_In: input port simulating HTTP traffic
FTP_In: input port simulating FTP traffic
TelNet_In: input port simulating TelNet traffic
SMTP_In: input port simulating mail protocols traffic
SNMP_In: input port to simulate the simple network management protocol.
ApplicationOut: output port to display received data
in: input port to receive data coming from the transport layer

X ∈ {HTTP data ∈ *N* , FTP data ∈ *N*, TelNet data ∈ *N*, SMTP data ∈ *N*, SNMP data ∈ *N*, transport layer data ∈ *N* };
S : {Sigma, X, Preparation Time}
Y ∈ { parsed application layer data ∈ *N* };
$\delta_{ext}$(s,e,x)
{
If passive
Case msg.port
HTTP_In
Identify protocol port and save data, signal output function to send to transport
FTP_In

```
                    Identify protocol port and save data, signal output function to send
                    to transport
                SMTP_In
                    Identify protocol port and save data, signal output function to send
                    to transport
                SNMP_In
                    Identify protocol port and save data, signal output function to send
                    to transport
                TelNet_In
                    Identify protocol port and save data, signal output function to send
                    to transport
                In
                    Signal output function to output to user
        Else
                continue
}
δint(s,e)
{
        Case phase
                active: passivate
                Default: continue
}
λ(s)
{
        Send application value to application out
}
```

### 3.1.1 The Transport layer

The second layer in the host is the transport layer. The layer is responsible for reliable, end-to-end data transfer through out the network. There are many protocols functioning in this layer, however as a starting point, TCP was chosen since it provides a solid base to build upon.

TCP model was broken into a set of two models; this was to facilitate full-duplex communications over the network. A complete overview of the model is shown below.

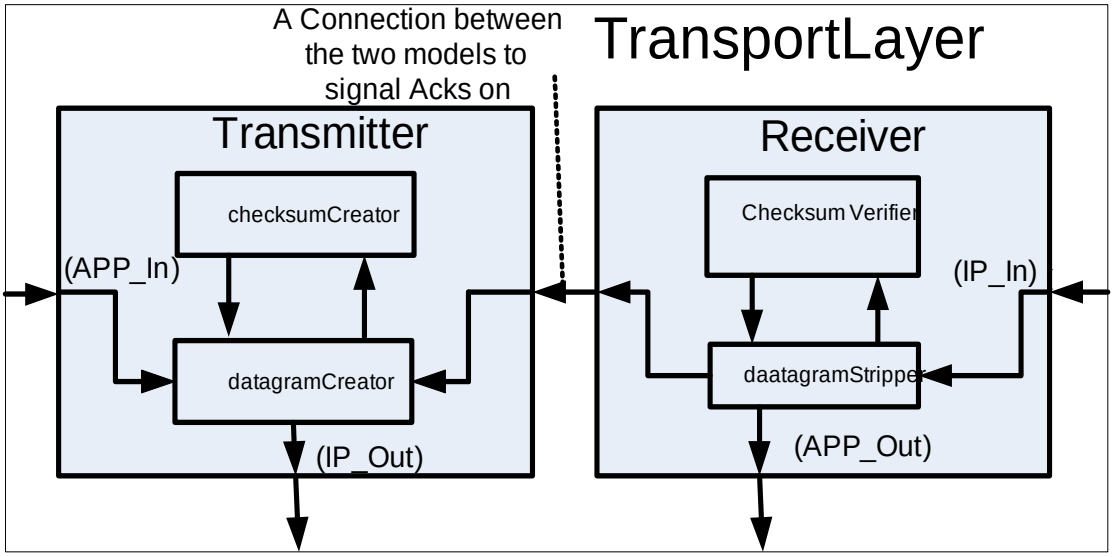


**Figure 16: TCP logical design**

The Transmitter module is responsible for receiving data from the application layer model; the data is then parsed in the format shown in figure 17, refereeing back to section 2.3 Traffic Format), it is seen that this packet is shown at the very bottom of the data sent out by the host models, before the Data Link Layer. This is to conform to the concept of layered protocols (where each layer builds on the one before it)

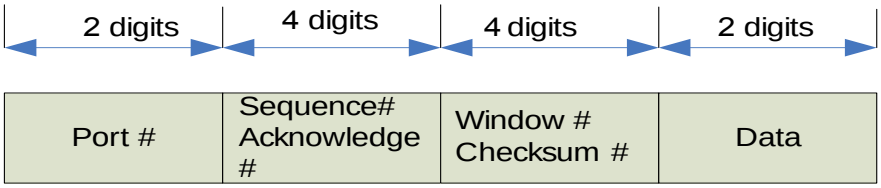


**Figure 17: TCP Packet format**

Parsing is done in steps, to accommodate the creation of the checksum. The event flow can be seen in the following diagram

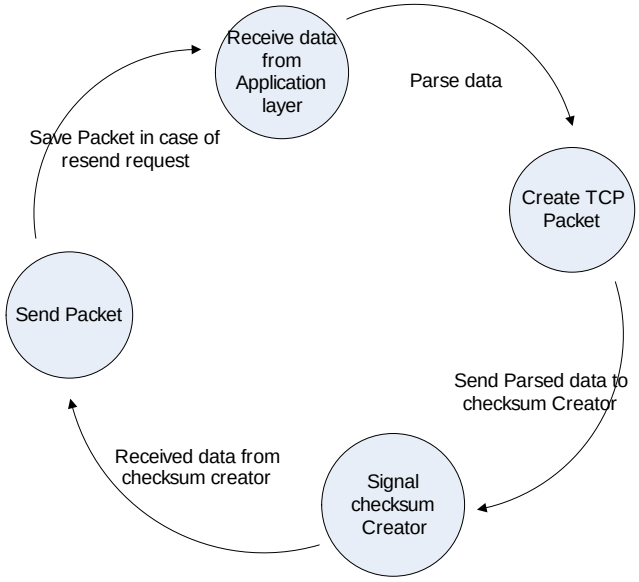


**Figure 18: Receiver Flow of events**

Parsing behaviour is split between two atomic models; "datagramCreator" and "checksumCreator". Data is received from the application layer in the "datagramCreator", the creator will create an initial packet and forward it to the "checksumCreator". The checksum creator will create the appropriate checksum, according to the received packet and forward the completed packet to the "datagramCreator". The packet is then sent to the next layer in the protocol stack. However before the packet is sent, it is saved in the model, to accommodate the connection manager, which will resend packets in case they are not received

The formal specification of the datagreamCreator is:

$$M=<I, X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D>$$

Where:

I(interface):

in: *general model input to receive application layer data on.*
Checkin: *input port to receive packet on after checksum has been created in the checksumCreator model*
ackPort: *input port to receive acknowledgments on from the datagram Stripper model*

ackSender: *an input port to receive requests to send acknowledgments on, the received data is the acknowledgment to send.*
gocheck: *an output port to send data (packet, with checksum field set to 0) to the checksum creator, to signal the model to create the checksum*
datagramCreatorOut*: model general output port to the network layer*

X *∈{ application layer data ∈ N , acknowldgment ∈ N, request to send ack ∈ N, parsed TCP packet with checksum ∈ N };*
S : {Sigma, X, Preparation Time}
Y *∈ { parsed TCP packet with checksum set to 0 ∈ N } ∪{ parsed packet to send ∈ N };*
$\delta_{int}$(s,e)
{
*Case phase*
*active: passivate*
*Default: continue*
}
$\delta_{ext}$(s,e,x)
{
*If passive*
*Case msg.port*
*In:*
*Create packet, and signal checksum creator to create a checksum*
*Checkin:*
*Packet received after checksum has been added, send to the network layer*
*ackPort:*
*check acknowledgement, to verify it is correct*
*if yes: delete saved packet*
*else: resend saved packet*
*ackSender:*
*send received data as an acknowledgment.*
}
λ(s)
{
*If received message is data from application layer, and checksum hasn't been created yet*
*Send packet with checksum field = 0 to checksum creator*
*If received message is data and checksum has been created*
*send data on the datagramOut port to the network port*
*If received message is an ack, check ack to be correct or not.*
*If not correct discard ack and resend the packet.*
*If received message is a request to send an ack*
*Send received message on the resend port to the network layer.*
}

And for the checksumCreatoer model:

$$M=<I, X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D>$$

Where:

I(interface):
In: *input port for the model to receive data, to create checksum on.*
checksumcreatorOut: *output port to send data with checksum value on, to the datagram creator, to be forwarded to the network layer.*
X *∈{ parsed TCP packet with checksum set to 0 ∈ N };*
S : {Sigma, X, Preparation Time}
Y *∈{ parsed TCP packet with checksum set ∈ N };*
$\delta_{int}$(s,e)
{
*Case phase*
*active: passivate*
*Default: continue*
}
$\delta_{ext}$ (s,e,x)
{
*If msg.port = in*
*Create the checksum*
}
λ(s)
{
*Send packet with checksum out on the checksumcretorOut model*
}

On the receiver side of the transport layer, a receiver module is used to receive data from the network layer. The module is made of two atomic models a "datgramStripper" and a "checksumValidator". The "datagramStripper" receives the data, from the network layer, which is also sent to the "checksumValidator". The validator will check the checksum field of the packet. If the checksum field is valid then the "datagramStripper" is notified, that the packet is not corrupted. Once the confirmation message is received the "datagramStripper" will check the packet type, to see if it is data or an acknowledgement. In case of data the packet headers are striped, and the data is forwarded to the application layer. The "datagramStripper" will also request the "datagramCreator" to send an Acknowledgment, to the source of the packet. On the other hand if the data is an Acknowledgement (data field is 0), the datagram stripper forwards the acknowledgment message to the datagramCreator to check if the acknowledgment is

expected, to either delete the saved packet or resend it. If the checksum is incorrect, the packet is simply discarded.

The datagramStripper formal specification is:

$$M=<I, X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D>$$

Where:

I(interface):

in: *an input pot to receive data coming from the network layer.*
Checkin: *an input port to receive confirmation of checksum on*
receiveAck: *an output port to send the datagram creator acknowledgments on.*
sendAck: *an output port to signal the datagram creator to send acknowledgments. The message sent from here is the ack.*
datagramstripperOut: *output port to the application layer.*

X *∈{ data from the network layer ∈ N , checksum validation ∈ N };*
S: *{Sigma, X, Preparation Time}*
Y *∈ {application data ∈ N } ∪{ request to send ack ∈ N } ∪{ ack signal ∈ boolean };*
$\delta_{int}$(s,e)
{
*Case phase*
*active: passivate*
*Default: continue*

}
$\delta_{ext}$(s,e,x)
{
*Case msg.port*
*In*
*Save msg.value() as received TCP data. Set send flag to false*
*Checkin*
*Check type of message*
*if data, and ack is correct set send flag to true to signal the output function to send the message to the application layer,*
*else if data and the ack is corrupted request resend of the packet*
*else if ack, forward the message to the datagramcreator model*
}
λ(s)
{
*If flag to send data to application layer is true*
*Send data to the application layer through the datagramStripper out port*
*If flag to send request for an ack*
*Send message to send to the datagram creator on the sendAck port*
*If flag that we received an ack is true*

*Send the acknowledgment received to the datagramCreator on the receiveAck port*

}

Methods used by the model to create the headers has the signature

**`Double datagramCreator::creator( double appData)`**

```
double datagramCreator::creator( double appData )
```

**Figure 19: Creator signature**

And the method used to create the checksum is

```
double checksumCreator::checksum( double Data )
```

**Figure 20: checksum method signature**

And for the checksumValidator model:

$$M=<I, X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D>$$

Where:

I(interface):

in: *input port to receive data from the network layer on*
checksumvalidatorOut: *output port to signal the datagramstripper model if the data is corrupted or not or if its an ack.*

$X \in \{ packetIn \in N , frameIn \in N, status \in N \};$
S : {Sigma, X, Preparation Time}
$Y \in \{ frameOut \in N \} \cup \{ packetOut \in N \} \cup \{ senseCarrier \in boolean \};$
$\delta_{int}(s, e)$
{
*Case phase*
*active: passivate*
*Default: continue*
}
$\delta_{ext}$ (s, e, x)
{
*save incoming message data, to verify the cheksum*
}
$\lambda(s)$
{

*Verfity the checksum*
*Send the result of the verification process to the datagramStripper through the checksumvalidatorOut port*
}

The method used to verify the checksum has the signature

```
int checksumValidator::validator (double Data)
```

**Figure 21: checksum Validator**

While the method used to strip the headers has the signature

```
double datagramStripper::stripper(double Data)
```

**Figure 22: Stripper method signature**

### 3.1.3 The Network layer

The third layer in the TCP/IP stack is the network layer. This layer is modeled by the Internet protocols. The network layer is responsible for end-to-end communication through out the network; it simulates a connection less network protocol, which is the Internet Protocol (IP).

The layer is divided into two coupled models; a receiver module and a transmitter module. The logical (Coupled model illustration) of the layer is shown below.

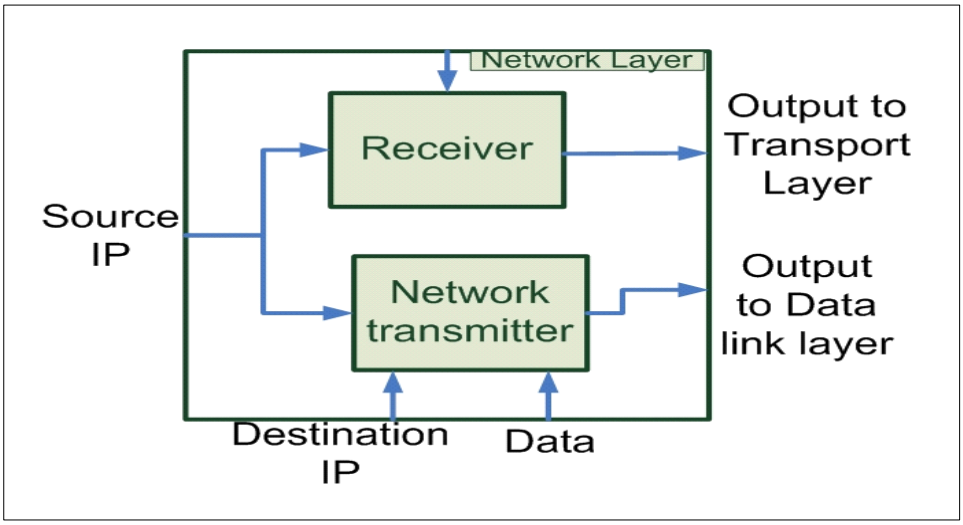


**Figure 23: IP logical design**

The models comprising the network layer are the network transmitter and the receiver. The network Transmitter receives data from the transport layer. The data is then parsed in the format illustrated earlier. The network transmitter would also save the destination IP in case of a resend request.

The model formal specification is:

$$M=<I, X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D>$$

Where:

I(interface):

ingress: *General Model input port to receive transport layer data on.*
resend: *port to receives resend requests on from the transport layer.*
SIP: *Input port to receive source IP on.*
DIP: *input port to receive destination IP (IP of machine we would like to send data to) on*
Egress: *output port to the data link layer.*

X ∈ { transport layer message to send ∈ *N* , request for resend ∈ *N*, source ip ∈ *N*, destination ip ∈ *N* };
S : *{Sigma, X, Preparation Time}*
Y ∈ { parsed Network layer data ∈ *N* };
$\delta_{int}$(s, e)
{
*Case phase*
*active: passivate*

*Default: continue*

}
$\delta_{ext}$(s, e, x)
{
*Case msg.port*
*SID*
*Save msg.value as source IP*
*DIP*
*Save msg.value as destination ip, and set resend value to msg.value.*
*Ingress*
*Create IP headers. Save msg.value as local value to send*
*Resend*
*Save msg.value() as local value to resend data*
*Set destination Ip to the resend ip value*
}
λ(s)
{
*If send*
*Send the four messages shown in section 2.3 Traffic Format) on the networkTransmitter egress port*
*If resend*
*Create the ip headers and send the received value through the networktransmitter output port.*
}

The network transmitter methods used are header maker, with the signature

```
double networkTransmitter::headerMaker()
```

**Figure 24: header maker signature**

The receiver's coupled model receives data from the Data Link layer and forwards it to the transport layer. The model removes all IP headers associated with the packet.

The model formal specification is as follows

$$M=<I, X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D>$$

Where:
I (interface):
Ingress: *General Model input port to receive data from the data link layer on.*
SIP: *Input port to receive source IP on.*
Egress: *General Output port to output data to the transport layer*

X ∈ { data link layer message ∈ *N* };
S : {Sigma, X, Preparation Time}
Y ∈ { message stripped of IP headers, to be sent to transport layer ∈ *N*};
$\delta_{int}$(s, e)
{
*Case phase*
*active: passivate*
*Default: continue*
}
$\delta_{ext}$ (s, e, x)
{
*Case msg.port*
*Ingress*
*Save msg.value as datalink layer data*
*SIP*
*Save msg.value as source ip*
}
λ(s)
{
*Strip data of headers, verify checksum*
*Send message on the receiver egress port to the transport layer.*
}

The receiver also makes use of the verifier method, with the signature

```
bool Receiver::verify (double Data)
```

**Figure 25: Verify method**

### 3.1.4 The Data link layer

Modeling the data link required dividing it into two parts; coding the CRC operations of the LLC sub-layer and modeling the CSMA/CD algorithm of the MAC sub-layer. However, the designs of both of these parts are combined into one atomic model called “dataLink”.

The DEVS specification of the ‘data Link’ model:

$$M=<I, X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D>$$

Where:

I: Model Interface,

getPacket: *receives packet from higher layer for transmission*
sendPacket: *sends a packet received from another device to the internet layer*
getFrame: *this port receives frames from another device via physical layer*
sendFrame: *sends a frame to the physical layer*
senseCarrier: *port connected to the physical layer model to sense its status*
status: *input port from physical model that indicates the status of carrier*

X ∈ { packetIn ∈ *N* , frameIn ∈ *N*, status ∈ *N* };
S : {Sigma, X, Preparation Time}
Y ∈ { frameOut ∈ *N* } ∪ { packetOut ∈ *N* } ∪ { senseCarrier ∈ boolean };
$\delta_{int}$: {
*if(carrier is busy)*
*send 0 at senseCarrier output port after 5 miliseconds*
*else*
*phase = passive;*
}
$\delta_{ext}$: {
*case port*
*getPacket:*
*case packet count*
*0: other.push_back(msg.value());*
*increment pcount;*
*1: destination.push_back(msg.value());*
*increment pcount;*
*2: source.push_back(msg.value());*
*increment pcount;*
*3: data.push_back(msg.value());*
*reset pcount;*
*sigma = preparationTime;*
*phase = active;*
*sense carrier is true;*
*getFrame:*
*case frame count*
*0: temp.other = msg.value();*
*increment fcount;*

```
                    1: temp.destination = msg.value();
                       increment fcount;
                    2: temp.source = msg.value();
                       increment fcount;
                    3: temp.data = msg.value();
                       increment fcount;
                    4: temp.fcs = msg.value();
                       reset fcount;
                       check for errors in frame using CRC function
                       if(no error)
                            sigma = preparationTime;
                            phase = active;
                            send packet is true;
            status:
                    if(status is idle)
                            if(there was a collision)
                                    resend previous frame sent
                            else
                                    if(frame was sent)  go to next element in queue
                                    send frame = true;
                            sigma = preparationTime;
                            phase = active;
                    if(status is a jam)
                            jamming is true;
                            sigma = preparationTime;
                            phase = active;
                    if(status is busy)
                            busy carrier is true;
                            sigma = preparationTime;
                            phase = active;
                    if(status is collision)
                            collision is true;
                            sigma = preparationTime;
                            phase = active;
    }
λ: {
    if(send frame)
            send all frame fields in the sendFrame output port
    if(send carrier)
            send 0 in the senseCarrier output port
    if(send packet)
            send all packet fields from the frame received into the sendFrame output
            port
    if(jamming is true)
            send 0 in the senseCarrier output port after random time
    if(collision)
```

*send -1 in the senseCarrier output port*

}

D: defined by the preparation time

The CRC operations are constructed in a header file (crc.h) included in the "datalink" atomic model. The CRC operations involve calculating the frame check sequence field before sending a frame, and detecting for errors when a frame is received. These operations are implemented similarly to the mathematical representation of the CRC-16 as shown earlier in this section.

The second part is modeling the carrier sense multiple access with collision detection algorithm. When a packet is received from higher-level protocol such as the 'networkTransmitter' model in the host, the CRC function appends a FCS field into the packet in order to create a frame. The frames that are ready to be sent are first pushed into a queue. Yet, before transmitting a frame the dataLink senses the carrier by sending a senseCarrier port message to the physical layer and waits for a response. Eventually, the physical layer would send its current status, which could be either one of the four: idle, busy, jammed or a collision. If the carrier were busy, the dataLink would send another senseCarrier message and wait for another response and it would repeat this process until the carrier is idle and then outputs the frames in the queue. However, after every frame sent, the dataLink sends a senseCarrier message to the physical layer again to ensure that from the status of the carrier there is no collision. If there was a collision the dataLink sends a jamming signal to the physical layer via the senseCarrier port with a message value of –1 and waits for a response from the carrier. The carrier responds by sending a jamming signal to all connected devices. Upon receiving this response the device that had their frames lost, will resend the frame that was stored in the queue after a random delay of 0 to 10 milliseconds. This random delay is determined by getting the first 'double' message of the frame that the device wishes to send and we divide that number by 10. The remainder is the random delay which is added to the msg.time() during output. In

contrast, if there was no collision after the frame was sent, the frame stored in the queue is deleted and the same scenario is applied for the next frame.

Besides sending frames into the physical layer, the dataLink model also receives frames sent by other devices through the physical layer. Upon receiving these frames, the frame is first checked for any errors by the cyclic redundancy check result. If there was no error the FCS field is stripped off the frame and the packet is sent directly to the network layer, namely the 'Receiver' model. The format of the frame sent and received over the physical layer is shown below. Each field corresponds to a single double variable and is sent sequentially.


**Figure 26: Frame message format**

### 3.1.5 The Physical layer

The last of the TCP stack is the physical layer. The physical layer is modeled as a simple atomic model, with the following logical view:

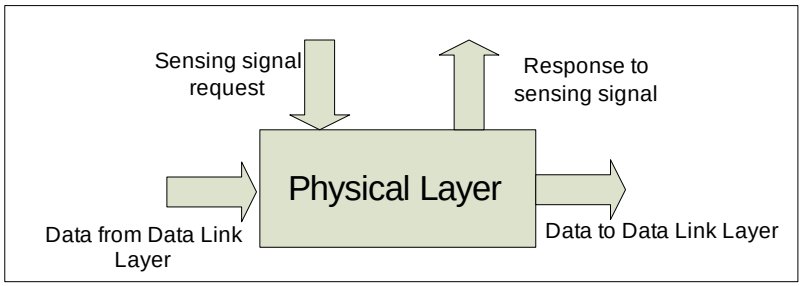


**Figure 27: Physical layer Logical view**

And its formal specification is

$$M=<I, X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D>$$

Where:
I(Interface)
in: *input port to receive data from physical layer end # 1*
in1: *input port to receive data from physical layer end # 2*
Out: *output port to output data to the physical layer end #1*
Out1: *output port to output data to the physical layer end # 2*
type: *an input port to set the the type of the link*
signal: *an output port to notify the data link layer of the current state of the link.*
sensingPort: *an input port to receive sensing requests from the data link layer on.*
X ∈ { request for sensing ∈ *N* , data to send on either ends of the link ∈ *N*, type of link∈ *N* };
S : {Sigma, X, Preparation Time}
Y ∈ { state of the link∈ *N* } ∪ { message passed through the link from one end to another ∈ *N* };
$\delta_{int}$(s, e)
{
*Case phase*
*active: passivate*
*Default: continue*

}
$\delta_{ext}$(s, e, x)
{
*Case msg.port*
*Type:*
*Save msg.value as link type*
*In:*
*Save msg.value as data to transfer, to other side (hence output on out1)*
*In1:*
*Save msg.value as data to transfer, to other side (hence output on out)*
*sensingPort:*
*check the state of the link.*
}
λ{s}
{
*If data from in, then output data on out1*
*If data from in1, then output data on out*
*If request for sensing, then output state on signal port*
}

### *3.2 Router*

we needed to model to model the router to connect network devises and segments together. To be able to simplify simulating the behavior of the router, we needed to take an abstracted look at the routing processes. In doing so we were able to abstract the router's behavior to three main functionalities; receiving and forwarding traffic, processing IP packets, and maintaining a routing table (discussed in section 2.2.2).

To simulate the three functions, three models were created; the interfaceingCard, the routerProcessor, and the RIPTable model. The router's model is shown with its three inner components in below in figure 20.

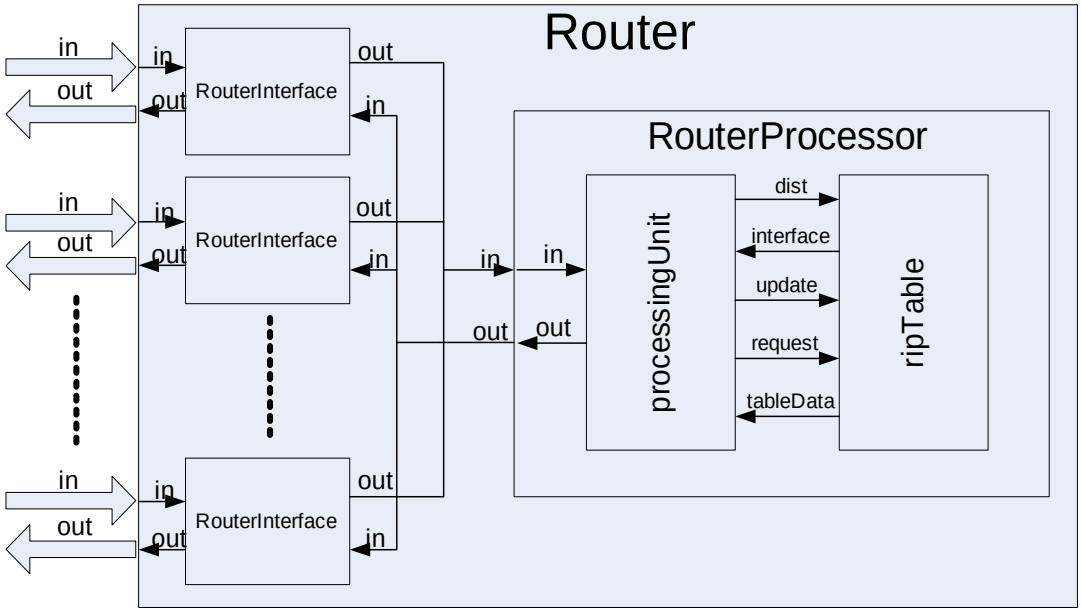


**Figure 28: the router's coupled model diagram**

The interfaceCard model will be responsible of receiving incoming traffic to the router and forwarding traffic out of the router. The routerProcessor model will take care of

processing the received packets, and the decision-making related to the packet. And the last model, the RIPTable, will maintain the router's routing table and will be responsible for accessing its data. Detailed description of the three models and their specifications will be provided in the following sections.

### 3.2.1 Router Interface

Every router has a number of interfacing cards or network cards that receive and forward traffic from and to the network. The number of interface cards that a router will have varies according to the router's design. The interface models that we have developed receive and send packets in the format discussed in section 2.3 of this report.

To handle traffic going in and out of the router through the interfaceCard model, we broke it into two separate atomic models; one for receiving packets from the network, and the other to forward packets out of the router to the network. The models are called RouterIn and RouterOut, respectively.

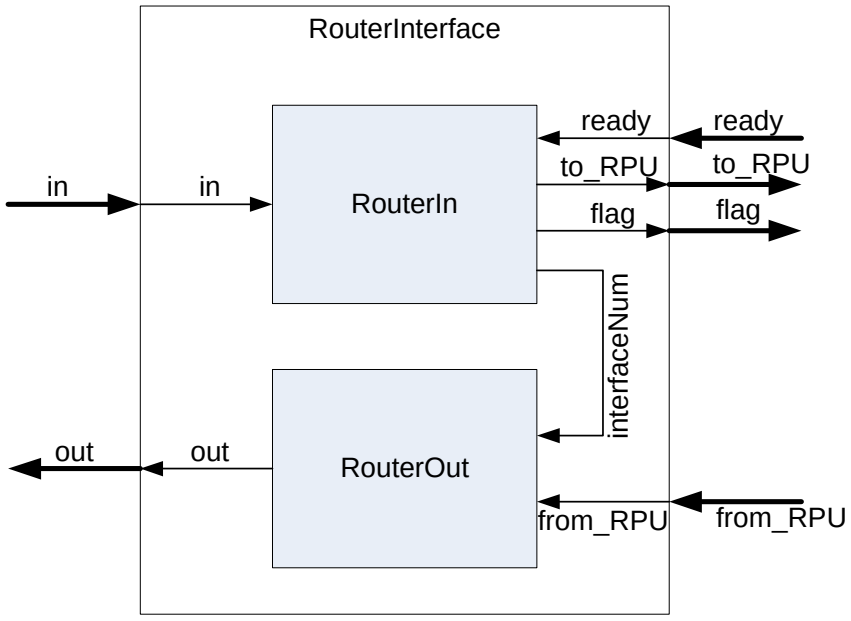


**Figure 29:The RouterInterface coupled model**

The RouterIn atomic model (shown in figure 20) was defined to receive the defined IP packets from the net and forward it to the RouterProcessor model. The model's specification is:

$$M=<I, X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D>$$

Where:

I:

in: *receives the IP packet.*
ready: *receives the router processor's signal requesting the ready packet.*
to_RPU: *for sending the packet to the router processor unit.*
flag: *to signal that a packet is ready.*
interfaceNum: *for sending the models ID number to the RouterOut model.*

X: $\in$ { IP packet $\in N$ , ready signal $\in N$ }
S: {Sigma, X, Preparation Time}
Y: $\in$ {IP packet $\in N$ } $\cup$ { flag $\in N$ } $\cup$ { ID $\in N$ }
$\delta_{int}$ (e, s):

```
{
    case phase:
        busy: passivat, receiving state = nothing, output_flag = 0.
        passive /* never happens */
}
```

$\delta_{ext}$(e,s,x):

```
{
    case msg.port:
        in:
            case receive state:
                nothing: store value1,set receiving state to got1, continue
                got1: store 2nd value, set receiving state to got2, continue
                got2: store 3rd value, set receiving state to got3, continue
                got3: store 4th value, outputFlage = 1, sigma= preparation,
                    S = active
        ready:
            prepare packet, outputFlag = 2, sigma = preparation, S =active
}
```

$\lambda$ (s):

```
{
    case outputFlage:
        0: send ID to RouterOut
        1: send flag to RouterProcessor
        2: send packet to RouterProcessor
```

}

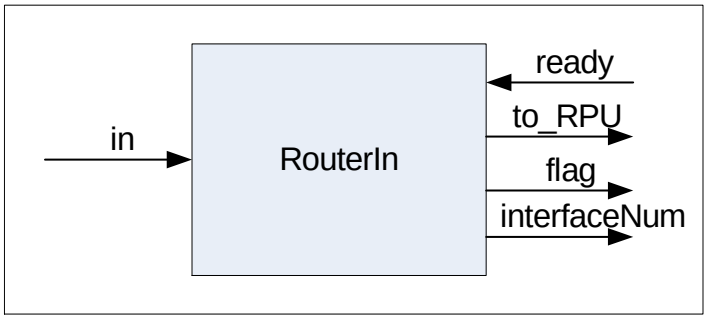


**Figure 30: the RouterIn atomic model**

The model has 2 input ports and 3 output ports to communicate with both the routers components and the network components. The "in" port is used to receive packets from the net in its externalFunction using the state machine described in the following state diagram.

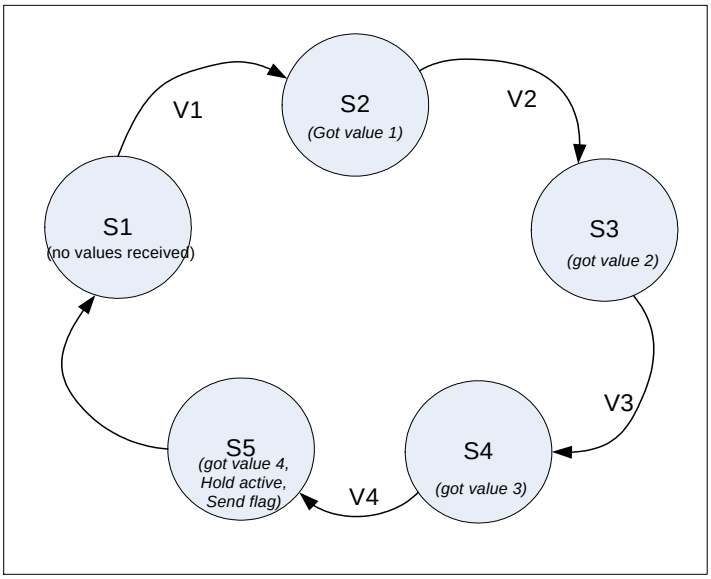


**Figure 31: packet receiving state machine**

Once the model is started, its receiving state is set to state S1 (received nothing). The model stays in this state until a packet starts to arrive at its input port "in". the first value of the packet triggers the transition V1 from state S1 to state S2 where the value gets stored and the model held in its passive state. The second value of the packet triggers V2 going from state S2 to S3 and the second value is stored. The same happens with the 3rd value in state S4. when the last value in the packet arrives at port "in", the transition V4 happens and we enter the state S5. In this state the model stores the last packet value, create a single packet using the 4 received valued, and stores the packet in its internal queue. In S5 the model sets is atomic state to active for the period of its preparation time which causes the model to execute its outputFunction and sends a flag signal to the RouterProcessor, notifying it that a packet has been received and is ready for processing. After flagging the RouterProcessor, the receiving state is set back to its default state S1.

The models second input port is called "ready". This port is used by the model to receive ready signals from the RouterProcessor as a notification that it is ready to processes the received packet. Since the ready signal is sent to all the routers interfaces, the interface will check the ready signal's value, and compare it to it ID. Only the interface with the ID matching to the ready signal will respond by forwarding its packet for processing.

As for the models output ports, each model has three of them, "flag", "to_RPU", and "interfaceNum". The "flag" port is used to output the flag signal to the RouterProcessor model with a value equal to the routerIn model's processes ID. The model's processes ID is a unique ID assigned to each model upon starting the simulation, and can be obtained by simply calling the function **id().** This flag signal is used by the RouterProcessor to order the packet for processing.

To forward the packet out of this model to the RouterProcessor model, the "to_RPU" output port is defined. The RPU in the ports name stands for the Router Processing Unit, which refers to the RouterProcessor model. Through this port, the routerIn model sends the packet from its internal queue to the RouterProcessor. the

packet is taken out of the model's internal queue and sent as received in four successive messages.

The last port in the model is the "interfaceNum" output port. The port is exclusively used to send the model's ID to the RouterOut model once the simulation starts. We wanted to send the models ID number to the RouterOut model, so the two models can use the same ID value since together they make the RouterInterface model. Having a unified ID number for the two components of the RouterInterface model enables as of keeping track of where each network is connected. The ID, which is used in flagging the RouterProcessor, becomes a tag for the packet the as long as it is in the router. This tag helps the router in identifying the port that it needs to replay to in case of a request packet, or the output interface number that should be stored in the routing table if the packet turns out to be an update packet.

The second atomic model that makes up the RouterInterface model is the RouterOut. This model was designed to simply forward packets out of the router to the network.

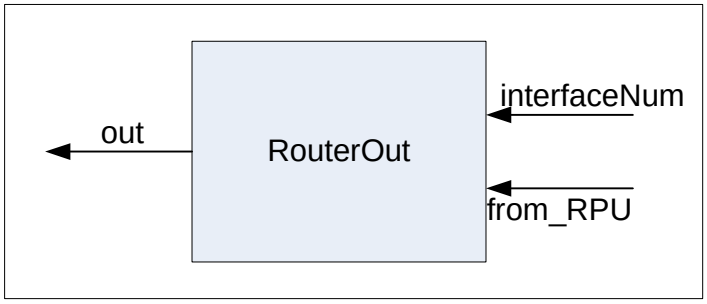


**Figure 32: the RouterOut atomic model**

The RouterOut specification is:

$$M=<I, X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D>$$

Where:

I:

interfaceNum: *receives the interface ID.*

From_RPU: *receives the packet to be sent out of the router.*
out: *for sending the packet.*

X: ∈ { forward data from router processor ∈ *N* , interface ID ∈ *N* }
S: {Sigma, X, Preparation Time}
Y: ∈ {IP packet ∈ *N* }
$\delta_{int}$ (e, s):
{
*case phase:*
*busy: passivat, receiving state = nothing.*
*passive /* never happens */*
}
$\delta_{ext}$(e,s,x):
{
*case msg.port:*
*in:*
*case receive state:*
*nothing: store value1,set receiving state to got1, continue*
*got1: store 2nd value, set receiving state to got2, continue*
*got2: store 3rd value, set receiving state to got3, continue*
*got3: store 4th value, receiveState = needPortNum, continue*
*needPortNum:*
*if msg.value = ID: sigma= preparation, S = active*
*else: receiveState = gotNothing, continue.*
*interfaceNum:*
*ID = msg.value.continue.*
}

λ (s):
{
*output the packet.*
}

as seen in the model's specifications, three ports were defined; "interfaceNum", "from_RPU", and "out". The "interfaceNum" port is the input port that gets connected with the RouterIn model to receive the interface number (the ID) that the model will use to identify its self. Since this ID is not the actual ID number for this model –which can be obtained by calling the function id()- , we will simply store the received ID value in an attribute and refer to it when needed. This way we can guarantee that both components making out the RouterInterface model will respond to the same Id number.

The "from_RPU" input port is the port that is the one responsible of receiving the packets from the RouterProcessor model. The receiving of a packet is handled the in a six state mechanism as shown in figure 25. The first five states and their activities are the same as described in the RouterIn model's state diagram and the only difference is in the last state S6. In S6 the models receive a value indicating the interface that should output the packet. We added this value to the set of outputs from the RouterProcessor, due to the desire of making the adding of new interfaces to the router as generic as possible. This last value enables us to add interfaces to the RouterProcessor without having to modify the source code of the RouterProcessor's models to add a new set of input and output ports corresponding to every new interface we add. It also allows the RouterProcessor to send a message through more than one interface at the same time in the case a broadcast of a request is needed. In this state S6, and after receiving the last signal from the RouterProcessor (the ID value), the model checks the received ID to see if the packet is to be forwarded or dropped. We designed the RouterOut model so it will output the packet only if the last message value is equal to its own ID, or if it is a negative value that does not equal the negative of its own ID. The last condition was added in particular to exclude an interface from a broadcasted update, in case the update happed due to information coming through that particular port. This technique is known as the Poisoned return, and is discussed in RIP2 (RFC # 2453).

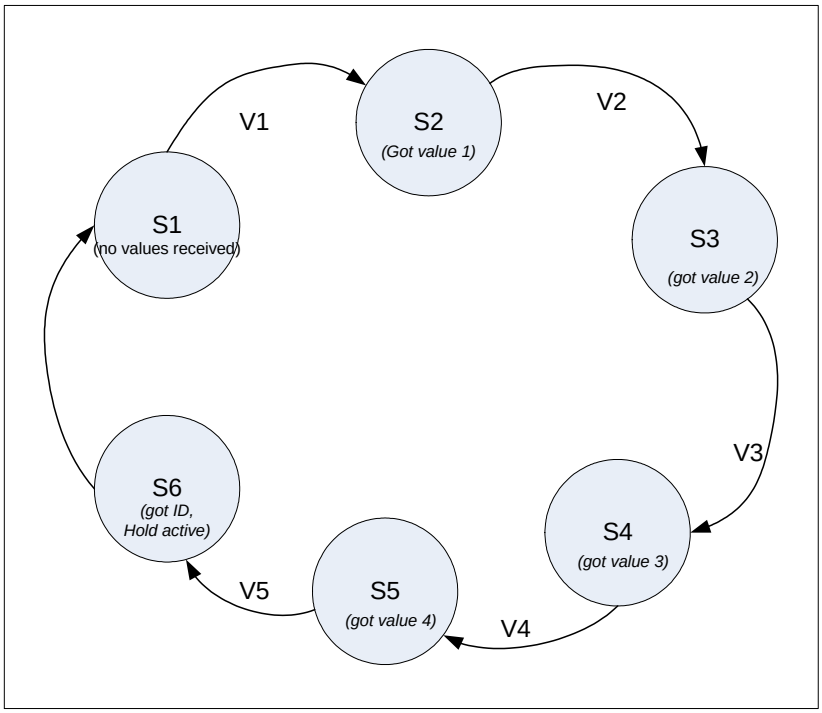


**Figure 33: receiving packets state diagram for the RouterOut model**

The only output port that this model have is the "out". It simply used to output the packet in its defined format once the ID value's was checked and the interface was selected for forwarding the packet.

### 3.2.2 Router Processor

After packet are received by the RouterInterface, they need to be processed to see if they are messages to the router (requests or updates), or if they are just data packets that needs to be forwarded to their destinations. To be able to handle the incoming packets, the RouterProcessor model was broken to two Atomic models, a queue and a PacketProcessor model.

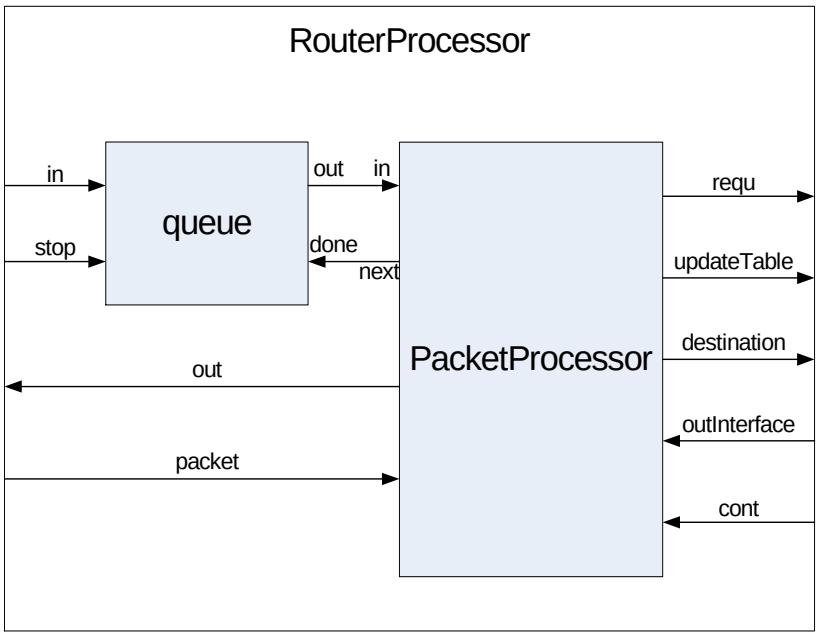


**Figure 34: the RouterProcessor model diagram**

The queue is a temporary storage device that uses the FIFO (first in first out) mechanism. This particular model was not created by any of the group members, but was adopted since its functionality is required for out project. The queue logical design is shown in the following figure:

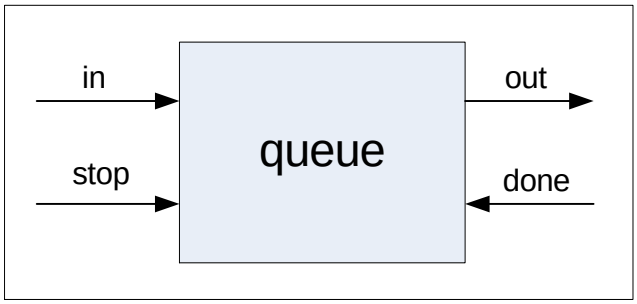


**Figure 35: the queue atomic model**

The queue is an atomic model that follows the DEVS formalism. It defines an input port "in" and an output port "out" to receive values that will be stored, and to output the data at a later time. To so, to extra input ports were defined in the model; "done" and "stop". Port "done" is used to signal the queue that the sent value was received, and that the model is asking for the next value in the queue. Upon receiving a signal on the "done"

port, the model eliminates the last sent element from its queue, and prepares and sends the next element (if any exists). The "stop" port is used as a regulator, a stop_send port. Using this port will trigger the model to output elements from its queue upon receiving requests, or simply ignoring requests and just storing received values until the model is toggled again by a signal on the "stop" port.

The queue model was used in the RouterProcessor to hold flag signals coming from the routers different interfaces. This is required, since multiple packets can arrive at different interfaces while the RouterProcessor is busy. Using the queue prevents losing any of the flag signals sent by the interfaceses, and allows the RouterProcessor to process packets in the order they arrived in at the different interfaces.

The PacketProcessor model represents the heart of the RouterProcessor. This model will be responsible for reading in the packets from the interfaces, processing the packets, and making routing decisions regarding their destinations.

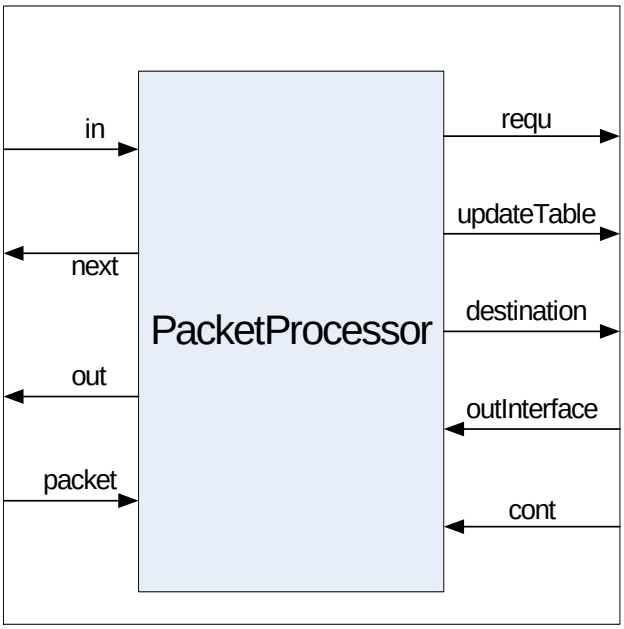


**Figure 36: the PacketProcessor atomic model**

The PacketProcessor specification is:

$M=<I, X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D>$

Where:

I:

in: *receives the interface ID from queue.*
packet: *receives the packet.*
outInterface: *to get the interface from the routing table*
cont: *get a signal to continue from the table*
next: *get next ID from queue*
getPacket: *request packet from interface*
requ: *send request data to table*
updateTable: *send update data to table*
destination: *send address to get output interface*
out: *for sending the packet.*

X: ∈ { flag ∈ *N* , output interface ID ∈ *N* , packet ∈ *N* , continue signal ∈ *N* }
S: {Sigma, X, Preparation Time}
Y: ∈ {IP packet ∈ *N* } ∪ { request data ∈ *N* } ∪ { update data∈ *N* } ∪ { destination address ∈ *N* } ∪ { next ∈ *N* } ∪ { get packet signal∈ *N* }
$\delta_{int}$ (e, s):

{
*case phase:*
*active: passivat, reset receiving state and output state.*
*passive /* never happens */*
}

$\delta_{ext}$(e,s,x):

{
*case msg.port:*
*in: get the packet from interface*
*packet:*
*case receive state:*
*nothing: store value1,set receiving state to got1, continue*
*got1: store 2nd value, set receiving state to got2, continue*
*got2: store 3rd value, set receiving state to got3, continue*
*got3: store 4th value, receiveState = needPortNum, outState*
*if packet data: outState = data.*
*If packet request: outState =request.*
*If packet update: outState = respond.*
*sigma= preparation, S = active*

*outputInterface:*
*outID = msg.value, outState = forward, sigma= preparation, S = active*
*cont: signal for next packet*
*}*

λ (s):
{
*case outState:*
*data: send destianation to get out interface.*
*Request: send requist data to table.*
*Update: send update information to table*
*Forward: send packet.*
}

The PacketProcessor starts its packet processing cycle upon receiving an event on its “in” input port. The received event will carry the flag value sent by the RouterInterface and stored in the queue model. Once the value is received the, the PacketProcessor will send it through its “get_packet” output port, that is connected to the “ready” input port of all the interfaces. Only the interface with the matching ID number to the sent value –as discussed previously in the router’s interface section (3.2.1)-, will respond by outputting the ready packet to RouterProcessor model, which in turn ends up at the PacketProcessor model. The packet processor receives the packet through its “packet” input port, using the same state mechanism used by the RouterIn model to receive packets.

After receiving the last value of the packet, the Packetprocessor checks the three most significant digits of the first value to determine its type. If the value is 1 then the packet is a request packet, if it is 2 the it is an update packet, other than that the packet is a data packet and the value represents the packet identification number (see section 2.3 for packet format). If the packet turns out to be a request packet (type 1), the PacketProcessor will extract the address field from the packet and forward it along with the ID of the interface that the packet arrived on through its “requ” port to the ripTable model. This data will be used by the ripTable model to respond to the request, as will be explained later in the Router Table section (section 3.2.3). The PacketProcessor will wait until it receives a confirmation from the ripTable model on its “cont” port, telling it that the request was answered and it can go ahead and get the next packet for processing.

The second type of packets received by the PacketProcessor is the update packets (type 2). For this packet type, the model will forward three values to the ripTable model through its “updateTable” port. The values that are forwarded are the interface ID, the destination address, and the metric value associated with that destination. The metric value is the three least significant digits in the first value of the packet (see section 2.3) After preparing the three values, the model holds it self in the active state for the predefined preparation time, and then executes the output function and send the three values –the address, followed by the metric then the interface ID number- using the “updateTable” output port. The model then - like in the case of a request packet-, waits for a signal from the ripTable model on its "cont" port before it request the next value in the queue.

The last type is data packets. For this type of packets, the PacketProcessor sends the destination address of the packet to the ripTable through the "address" port, requesting the number of the interface that will be used to forward the packet. The model then waits for the interface number on its "outInterface" port, and then forwards the packet followed by the interface ID number to the RouterInterface models. After receivint the output interface number, the packet is forwarded by sending the packets values followed by the interface number through its "out" port to the RouterInterface models.

To indicate that the destination address was not found in the routing table,The value 0 was assigned as a special signal from the ripTable model to the PacketProcessor. When the PacketProcessor sees this value, it issues a request packet through all interfaces except the one that the packet was received through, requesting an update on that destination. The PacketProcessor uses the "out" packet to send the request.

### 3.2.3 Router Table

The model's main functionality is to maintain the routing information that the router needs to forward packets to there destinations. The simple format that we used for entries in the table is:

| Address | Metric | Interface |
|---|---|---|

**Figure 37: Router table entry format**

Where the address is destination that the packet wants to get to, the metric is a value that represents the cost of getting to that destination, and the output interface is the number of the interface that the router must forward the packet through to get it to its destination or at least one hop closer in the right direction.

According to the RIP2 protocol, the routing table can have other entries in the table such as the subnet mask and a set of flags associated with each entry in the table. In the RIP2 protocol RFC, it is stated that the router table must send updates to its neighbor routers every 30sec. Although the CD++ uses a time value with its messages, the time doesn't represent real time values that can be used to set delays or trigger an action with a predefined time value. For that reason, we could not set a timer for the updates, and updates are not sent every 30sec. this functionality can be implemented using the real time version of the tool, and is left for future work.

To compensate for not sending periodic updates, and for keeping all routers updated on various topology changes occurring. We had the table model send an update packet every time it updates its routing table. When the model sends an update, it forwards it through all interface ports, except the one it who originated the update, this is to simulate poisened return criteria to control amount of traffic dedicated to router updates.

The ripTable model was created as a single atomic model. The model will be receiving events on one of its 3 input ports; address, update, or request. Each port of the three will be receiving events related to a specific packet type. Depending on the event the model will send out events using its output ports "out", "outInterface", and "done". The model is shown below.

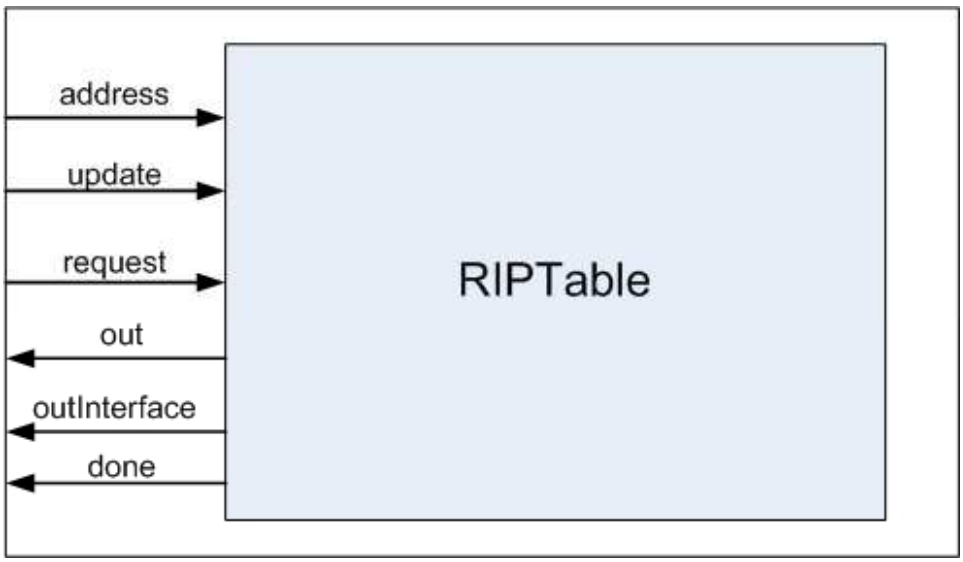


**Figure 38: the RIPTable atomic model**

The model's specification is:

$$M=<I, X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D>$$

Where:

I:

address: *receives the destination address.*
updtae: *receives the update data.*
request: *receives request data*.
done: *signal end of operation*
outInterface: *sends the output interface for the packet*
out: *for sending response packets.*

X: $\in$ { destination address $\in N$ , update data ID $\in N$ , request data $\in N$ }
S: {Sigma, X, Preparation Time}
Y: $\in$ {response packets $\in N$ } $\cup$ { output interace $\in N$ } $\cup$ { done signale $\in N$ }
$\delta_{int}$ (e, s):

{

*case phase:*

*active: passivat, reset receiving state and output state.*

```
                passive /* never happens */
    }
δext(e,s,x):
    {
        case msg.port:
                address: get the packet from interface, outState = forwardPort, sigma=
                        preparation, S = active
                update:  receive update data.
                        If we update table, sigma = preparation, outState = respond_1, S
                        = active
                        If no update needed, continue.

                request:
`                       get request data,
                                if address filed = address, outState respond_1,
                                else if address == 0, outState = respond_all
                        sigma= preparation, S = active
    }

λ (s):
    {
        case outState:
                forwardPort: send outinterface from table.
                Respond_1: send 1 update packet.
                Respond_all: send all table entries.
    }
```

The "address" is the port that the model will receive the packets destination address on, from the router processor. After receiving the address, the model will search through its table for the received address. After the table look up, the model outputs the interface that should be used by the RouterProcessor to forward the packet on, this message is outputted through the “outInterface” port.

In case the received destination is not found in the table, the model will send the value '0' instead of the interface ID, which will be handled by the processor by requesting an update on that address.

Updates are passed to the table through the "update" port. The updates are received as three values; the address, the metric, and the interface ID. Once the three values are received, the table will iterate through its entries looking for the address and if

not found, the new information is added to the table. If the table already contained an entry for this destination address, the table will compare that entry's metric value with the new received metric after incrementing it by 1. The increment is done to follow the RIP standards where it says that for every hop adds 1 to the metric up to a value of 16 – as 16 is the maximum number of hops allowed-. If the new metric value is less that the existing one in the table, the old entry is removed and the table is updated with the new values. After updating the table, a packet is sent out trough the "out" port to update adjacent routers of the new change. The update is sent through all interfaces except the one that the update message arrived from. Following the update signal, the model sends a confirmation to the routers processor using its "done" port.

The last set of events is requests for updates, which arrives on the models "request" port. The request message is two values carrying the address that the requesting router wants to be updated about, and the interface to be used to forward the update information through. The address can be either an IP address, or the value 0. If the value 0 was used, then the table will send all of its information as updates as the RIP2 protocol states. The tables information will be sent as update packets through the model's "out" port, and as in the case of receiving updates, the model will send a signal to the router's processor on the "done" port to indicate that it has finished its work.

We have also included a more complete version of Dijkstra in Appendix “X” the essence of the algorithm is stated as follows, “*Given a network of nodes connected by bidirectional links, where each link has a cost associated with it in each direction, define the cost of a path between two nodes as the sum of the costs of the links traversed. For each pair of nodes, find the path with the least cost*” [13:342].

### *3.3 Hub*

The hub is a simple atomic model; it has also served as our test bed for converting the library models into a parallel environment. The model logical design is as follows

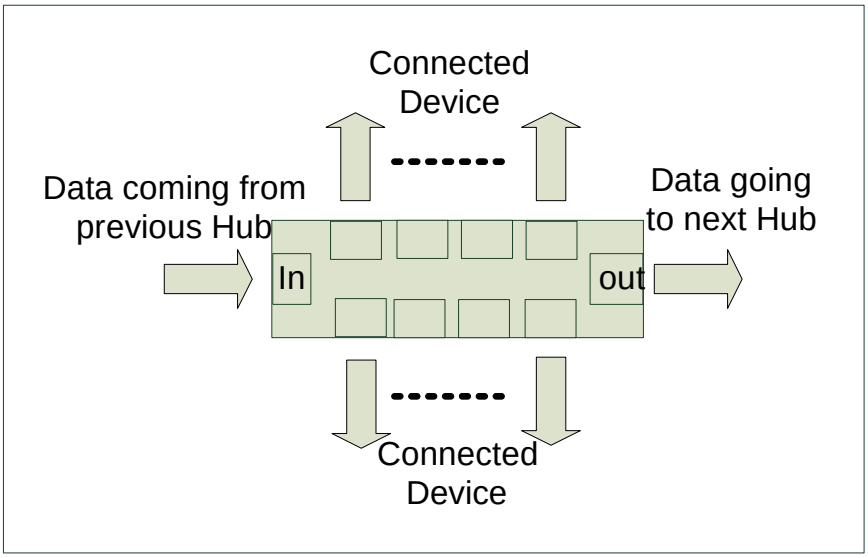


**Figure 39: Hub Logical design**

The Hub formal specification is

I(interface):
Ingress: *input port to receive data from interconnected devices on*
Setter: *input port to set hub specific information from*
Egress1: *output port to 1st connected device*
Egress2: *output port to 2nd connected device*
Egress3: *output port to 3rd connected device*
Egress4: *output port to 4th connected device*
Egress5: *output port to 5th connected device*
Egress6: *output port to 6th connected device*
Egress7: *output port to 7th connected device*
Egress8: *output port to 8th connected device*
Egress9: *output port to inter-networking device*

X: ∈ { data ∈ $N$ , setting information ∈ $N$ };
S: {Sigma, X, Preparation Time}
Y: ∈ {regenerated data ∈ $N$ };
$\delta_{int}$ (e, s):
{

```
        case phase:
                active: passivat
                passive /* never happens */
    }
δext(s, e, x):
    {
        case msg.port:
                ingress: set localvalue to msg.value
                setter: set local data field (hub identifier) to msg.value
}

λ (s):
{
        Output data to all egrees ports
}
```

# 4 Testing

In order to show the functionality of the library models, we have included some of the major tests, concerning the more complicated models worked on. This chapter will walkthrough most of the models in comprising the library device. Model, event, log and out files for all the tests are included in Appendix "X".

## *4.1 Host model*

The host testing section aims at proving that the models comprising the host are functioning correctly. The remaining of the section will trace data originating at one point to the end of the host models.

### 4.1.1 Application layer

The application layer is a simple atomic model, hence chosen to acquaint the reader with the terms and methodology of testing. The functionality of the layer is to parse the data, with the port value and forward the newly created value to the transport layer.

The following section of the event file shows data inputted from various ports simulating various applications interacting with the simulation.

```
00:00:15:000 outtotransport 1180  //Application data sent on HTTP Port
00:00:21:000 outtotransport 1280
00:00:27:000 outtotransport 1380
00:00:33:000 outtotransport 1480
00:00:40:000 outtotransport 1580
.
00:02:55:000 outtotransport 1125   // Application data sent on Port 25
00:03:05:000 outtotransport 1325
00:03:15:000 outtotransport 1425
00:03:25:000 outtotransport 1525
```

**Figure 40: Application Output file**

The values are outputted, after the time advance function defined in the model file elapses

```
[application]
preparation : 00:00:05:000
```

**Figure 41: preparation Time**

The model's event file shows the inputed data to be

```
00:00:10:00 infromHTTPuser 11   // data inputted on HTTP input port
00:00:16:00 infromHTTPuser 12
00:00:22:00 infromHTTPuser 13
00:00:28:00 infromHTTPuser 14
00:00:35:00 infromHTTPuser 15
.
00:02:50:00 infromSMTPuser 11
00:02:60:00 infromSMTPuser 12
00:03:00:00 infromSMTPuser 13
00:03:10:00 infromSMTPuser 14
00:03:20:00 infromSMTPuser 15
```

**Figure 42: Application event file**

The event file shows the discrete inputs, while the output file illustrated earlier shows the output from the layer.

### 4.1.2 Transport layer

Data outputted from the application layer is received by the transport layer. Thus the following event file for the transport layer shows inputs in the output format of the application layer.

```
00:00:10:00 infromApplication 1280
00:00:16:00 infromApplication 1280
00:01:22:00 infromApplication 1380
00:02:28:00 infromApplication 1480
00:03:35:00 infromApplication 1580
```

**Figure 43: Transport layer event file**

The event file shows data coming from an HTTP port, with different values. The transport layer responds also by parsing the data in the appropriate format discussed in section 2.3 (Header format).

```
00:00:10:015 outtonetwork 1.2e+12
00:00:16:015 outtonetwork 1.2e+12
00:01:22:015 outtonetwork 1.3e+12
00:02:28:015 outtonetwork 1.4e+12
00:03:35:015 outtonetwork 1.5e+12
```

**Figure 44: Transport layer output**

The transport layer output, doesn't give mush explanation to what happened, due to deficiencies in the tool, however the log file shows the exact values outputted by the model, this is illustrated below

```
Mensaje X / 00:00:10:000 / Root(00) / infromapplication /   1280.00000
para top(01)
Mensaje X / 00:00:10:000 / top(01) / in /   1280.00000 para
datagramcreator1(02)
Mensaje D / 00:00:10:000 / datagramcreator1(02) / 00:00:00:005 para
top(01)
Mensaje D / 00:00:10:000 / top(01) / 00:00:00:005 para Root(00)
Mensaje * / 00:00:10:005 / Root(00) para top(01)
Mensaje * / 00:00:10:005 / top(01) para datagramcreator1(02)
Mensaje Y / 00:00:10:005 / datagramcreator1(02) / gocheck /
1200000000080.00000 para top(01)
Mensaje D / 00:00:10:005 / datagramcreator1(02) / ... para top(01)
Mensaje X / 00:00:10:005 / top(01) / in / 1200000000080.00000 para
checksumcreator1(04)
Mensaje D / 00:00:10:005 / checksumcreator1(04) / 00:00:00:005 para
top(01)
Mensaje D / 00:00:10:005 / top(01) / 00:00:00:005 para Root(00)
Mensaje * / 00:00:10:010 / Root(00) para top(01)
Mensaje * / 00:00:10:010 / top(01) para checksumcreator1(04)
Mensaje Y / 00:00:10:010 / checksumcreator1(04) / checksumcreatorout /
120000000 9280.00000 para top(01)
Mensaje D / 00:00:10:010 / checksumcreator1(04) / ... para top(01)
Mensaje X / 00:00:10:010 / top(01) / checkin / 1200000009280.00000 para
datagramcreator1(02)
Mensaje D / 00:00:10:010 / datagramcreator1(02) / 00:00:00:005 para
top(01)
Mensaje D / 00:00:10:010 / top(01) / 00:00:00:005 para Root(00)
Mensaje * / 00:00:10:015 / Root(00) para top(01)
Mensaje * / 00:00:10:015 / top(01) para datagramcreator1(02)
Mensaje Y / 00:00:10:015 / datagramcreator1(02) / datagramcreatorout /
1200000009280.00000 para top(01)
```

**Figure 45: Transport layer log file**

The log file shows at time "00:00:10:000" inputted data to the transport layer. The data is then passed down to the "datagramCreator" through the top. At time

"00:00:10:005" the datagramcCreator sends the initial packet created, to the checksum creator model. The checksum creator responds at time "00:00:10:010" with the same packet sent to it, with the addition of the checksum values, highlighted in green. The data is then returned back to the datagramCreator through the checkin port highlighted in yellow, where it is sent to the network layer through the datagramCreatorOut port at time "00:00:10:015". The output file shows the first two digits of the data being sent.

### 4.1.3 Network layer

The third layer is the network layer. This layer carries the information of the sending party as well as the destination; Since the IP is a connection-less protocol. Addressing information, is supplied to the models through the event file as such

```
00:00:00:010 infromTransport 1122334455580  // data to send
00:00:00:020 DestinationIP 192168111223     // destination IP value
```

**Figure 46: Network Layer event file**

It should be noted that the layer did not get a source IP; this is because we have sent the IP through the model file. This was done to facilitate creating a tool that would automatically create host units

```
[networkTransmitter1]
IP : 111222333
```

**Figure 47: Source IP field**

The information sent to the network layer is used to create a checksum value, which is used to verify the data sent over the network

The model outputs the required four fields specified in section 2.3 traffic format

```
Mensaje Y / 00:00:13:020 / networktransmitter1(02) / egress /
485000015500.00000 para top(01)
Mensaje Y / 00:00:13:020 / networktransmitter1(02) / egress /
192168116224.00000 para top(01)
Mensaje Y / 00:00:13:020 / networktransmitter1(02) / egress /
192168116224.00000 para top(01)
Mensaje Y / 00:00:13:020 / networktransmitter1(02) / egress /
12223334318080.00000 para top(01)
```

**Figure 48: Network layer log file showing output**

Again the log file is used to show the values outputted by the model, since the output file only shoes portions of the actual data as seen in the following figure

```
00:00:13:020 outtodatalink 4.85e+11
00:00:13:020 outtodatalink 1.92168e+11
00:00:13:020 outtodatalink 1.92168e+11
00:00:13:020 outtodatalink 1.22233e+13
```

**Figure 49: network layer output file view**

### 4.1.4 Data link layer

In order to test the dataLink model, another atomic model called ‘test2’ was made that would interact with the dataLink as if it is the physical layer. The sole purpose of the ‘test2’ model is to send to the dataLink the different possibilities of the connection link status. However, frames sent and received are via the input and output interface of the simulator in order to ensure that the packets/frames are sent and received accordingly. The coupled model interface for testing is shown in figure 50.

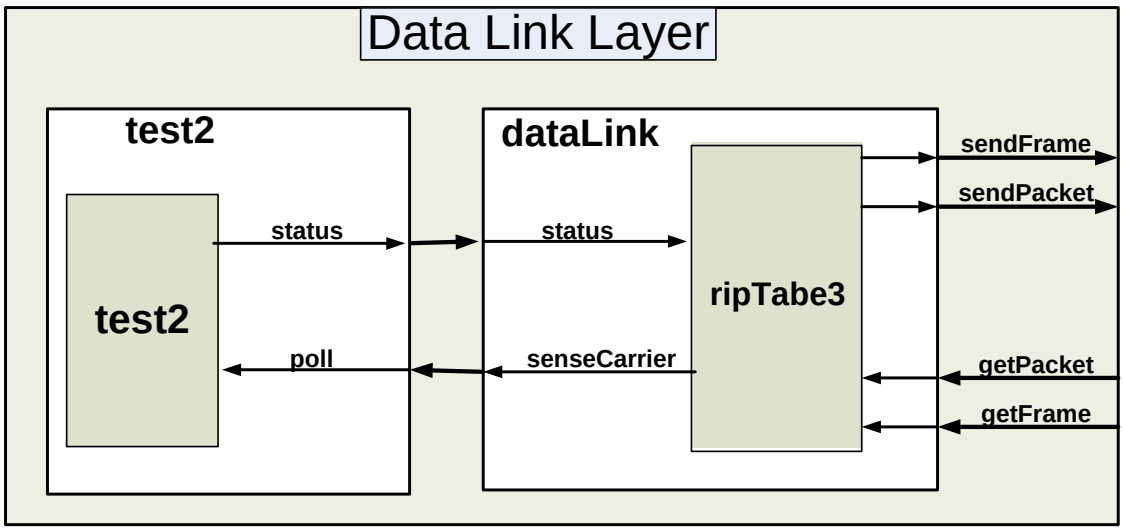


**Figure 50: Data Link ayer test model**

The event file is shown in figure 51, the simulated output is in figure 52. The event file tests the dataLink by sending it a frame with no errors and an IP packet. The output file displays the packet that was part of the frame received. The output file also shows displays the frame created when the packet was received at 20 seconds. Note that the preparation time was made 00:00:00:00 because there would be no processing delay so that there is a clear relationship between the input events and the output. Thus, the operation of the dataLink can be proven to be correct.

```
00:00:10:00 frameIn 101
00:00:10:00 frameIn 102
00:00:10:00 frameIn 103
00:00:10:00 frameIn 104
00:00:10:00 frameIn 410
00:00:20:00 packetIn 201
00:00:20:00 packetIn 202
00:00:20:00 packetIn 203
00:00:20:00 packetIn 204
```

**Figure 51: Data Link Layer event file**

```
00:00:10:000 packetout 101
00:00:10:000 packetout 102
00:00:10:000 packetout 103
00:00:10:000 packetout 104
00:00:20:000 frameout 201
00:00:20:000 frameout 202
00:00:20:000 frameout 203
00:00:20:000 frameout 204
00:00:20:000 frameout 810
```

Finally an integration test for all models comprising the host was made. The test will show the input of the IP through the model file, as follows

```
[networkTransmitter1]
ip : 111222333
```

**Figure 53: Host source IP**

This step was taken to facilitate building a tool that would create instances of the host model. The model receives user input through an event file, as follows

```
00:00:10:00 FTP_In 11
00:00:10:00 Destination 192168111
00:00:10:01 statusCarrier 1
00:00:40:02 FTP_In 1001214
00:00:40:02 Destination 192168001
00:00:40:03 statusCarrier 1
00:00:80:04 FTP_In 1001215
00:00:80:04 Destination 192168001
00:00:80:06 statusCarrier 1
00:01:90:07 Telnet_In 1001216
00:01:90:07 Destination 192168001
00:01:90:11 statusCarrier 1
```

**Figure 54: Host event file**

The event file shows FTP data from the host to another end on the network. Simple values where chosen here, so as to ease the process of reviewing the results.

The host reacted to the entries, shown in the event file, by creating the necessary headers, this is seen in the host output file.

```
00:00:25:000 sensecarrier 0
00:00:45:003 hout 6.72016e+08
```

```
00:00:45:003 hout 1.11222e+08
00:00:45:003 hout 1.92168e+08
```

**Figure 55: Host output file section**

However, as mentioned earlier, output files truncate the data, for that sections of the log file are shown here, to illustrate the host activities.

```
Mensaje Y / 00:00:49:010 / networktransmitter1(02) / egress /
2000000000.00000 para top(01)
Mensaje Y / 00:00:49:010 / networktransmitter1(02) / egress /
111222333.00000 para top(01)
Mensaje Y / 00:00:49:010 / networktransmitter1(02) / egress /
0.00000 para top(01)
Mensaje Y / 00:00:49:010 / networktransmitter1(02) / egress /
0.00000 para top(01)
Mensaje D / 00:00:49:010 / networktransmitter1(02) / ... para top(01)
Mensaje Y / 00:00:49:010 / top(01) / outtodatalink / 2000000000.00000
para Root(00)
Mensaje Y / 00:00:49:010 / top(01) / outtodatalink / 111222333.00000
para Root(00)
Mensaje Y / 00:00:49:010 / top(01) / outtodatalink /      0.00000 para
Root(00)
Mensaje Y / 00:00:49:010 / top(01) / outtodatalink /      0.00000 para
Root(00)
```

**Figure 56: host log file section**

This section of the host log file shows two events, the first being the host sending the received data, through out the network , after adding the appropriate headers, and that the datalink layer has actually responded as in the following figure

```
Mensaje Y / 00:00:06:000 / internet(09) / outtodatalink /  20000.00000
para top(01)
Mensaje Y / 00:00:06:000 / internet(09) / outtodatalink /
192168116224.00000 para top(01)
Mensaje Y / 00:00:06:000 / internet(09) / outtodatalink /      0.00000
para top(01)
Mensaje Y / 00:00:06:000 / internet(09) / outtodatalink /      0.00000
para top(01)
Mensaje D / 00:00:06:000 / internet(09) / ... para top(01)
Mensaje X / 00:00:06:000 / top(01) / getpacket /  20000.00000 para
datalink(02)
Mensaje X / 00:00:06:000 / top(01) / getpacket / 192168116224.00000
para datalink(02)
Mensaje X / 00:00:06:000 / top(01) / getpacket /      0.00000 para
datalink(02)
Mensaje X / 00:00:06:000 / top(01) / getpacket /      0.00000 para
datalink(02)
```

**Figure 57:log file illustrating data link interaction.**

Figure 57, shows the output of data from the network layer to the datalink layer, it also shows that the data link layer has actually stored the data, until it checks the physical layer. As the response arrives from the physical layer, data is sent to the other host.

## *4.2 Router model*

The router model is made of three separate models, two of which are coupled models. To test the router, we tested the atomic model that formulates the router and its components first, and then we tested the router as a whole.

### 4.2.1 Router's components tests

the router -when broken down- is made of five atomic models; the RouterIn and the RouterOut (constructing the RouterInterface coupled model), The queue and the PacketProcesor (constructing the RouterProcessor), and finally the RIPTable atomic model.

**Testing the RouterInterface model**

The testing process for this model was done through testing each one of its two components on its own. We started by testing the RouterIn model by creating a model file to link the models ports as defined by the DEVS formalism (see section 2.1 for the DEVS formalism and the CD++ tool). An event file was created in which multiple packets were sent to the model through its "in" port. The resulting output file from the simulation showed that the model behaved as required. A section of the event file that was used to test the model is shown below, along side with the corresponding section of the output file. A full cycle of receiving a packet and forwarding it to the RouterProcessor is highlighted in the attached file.

```
00:00:05:000  in  1.1          // sending the 1st packet
00:00:05:001  in  1.2
00:00:05:002  in  1.3
00:00:05:002  in  1.4
00:00:06:000  in  2.1          // sending the 2nd packet
00:00:06:001  in  2.2
00:00:06:002  in  2.3
00:00:06:003  in  2.4
00:00:07:000  ready  3         // requesting a packet from interfqace #3
00:00:08:000  ready  4         // requesting a packet from interfqace #4
00:00:09:000  ready  2         // requesting a packet from interfqace #2
00:00:09:100  ready  2         // requesting a packet from interfqace #2
```

**Figure 58: RouterIn event file**

```
00:00:00:010 interfacenum 2   // the interface send its ID
00:00:05:012 flag 2           // the 1st packet was recived
00:00:06:013 flag 2           // the 2nd packet was received
00:00:09:010 to_rpu 1.1     // send the 1st packet to the processor unit
00:00:09:010 to_rpu 1.2
00:00:09:010 to_rpu 1.3
00:00:09:010 to_rpu 1.4
00:00:09:110 to_rpu 2.1       // send the 2nd
00:00:09:110 to_rpu 2.2
00:00:09:110 to_rpu 2.3
00:00:09:110 to_rpu 2.4
```

**Figure 59: RouterOut output file**

We can see from the output file that the first message that the model sends is the models ID as discussed in the design. After that, as we inject packets from the event file, we see that the model will send its ID after receiving every packet to signal that a packet is ready for processing. The last 4 messages that was sent in the event files represent values that the interface might receive when the router's processor is asking for the ready packet. We can see from the message time in the output file that the model only outputs the packets when it receives its own ID.

As for the RouterOut model, another simulation was ran to test its functionality. An event file was created, as before, to feed messages to the model. The messages were a set of packets, each followed by a value representing the models ID. As with the RouterIn both the event and output file are shown below.

```
00:00:03:000 interfaceNum 32  // assigning ID 32 to the model
00:00:05:000 from_RPU 1.1     // sending 1st packet followed by ID 30
```

```
00:00:05:001 from_RPU 1.2
00:00:05:002 from_RPU 1.3
00:00:05:003 from_RPU 1.4
00:00:05:004 from_RPU 30
00:00:06:000 from_RPU 2.1     // sending 2nd packet followed by ID 32
00:00:06:001 from_RPU 2.2
00:00:06:002 from_RPU 2.3
00:00:06:003 from_RPU 2.4
00:00:06:004 from_RPU 32
```

**Figure 60: the RouterOut event file**

```
00:00:06:054 out 2.1          // outputting the 2nd packet
00:00:06:054 out 2.2
00:00:06:054 out 2.3
00:00:06:054 out 2.4
```

**Figure 61: the RouterOut output file**

From the output file shown in figure 61 we can see that the RouteOut outputs packets after checking the ID value sent after each packet, as desired in the design.

Both models were coupled and tested together, and the same results were obtained from their simulation since there is only one message send between the two models.

### Testing the RouterProcessor model

To test the model, we only need to test the PacketProcessor model (since the queue is part of the CD++ tool and was tested before). To do so, the model file "PacketProcessor.ma" was created, and a simulation was run using the event file "PacketProcessor.ev"in. The event file simulated the signal coming from the RouterProcessor's queue, and the packets that will be sent from the different interfaces. Packets with different types were fed to the PacketProcessor model and the output were analysed to check if the model behaved ion the desired fashion. Parts of the event file and the corresponding output file are shown below.

```
00:00:00:001  in  1              // send a flag signal
00:00:00:010  packet  3000000001 // send an data packet
00:00:00:010  packet  1.2
00:00:00:010  packet  1.3
```

```
00:00:00:010  packet  1.4
00:00:00:050  outInterface  2   // send output interface (as the table
respond)
00:00:01:001  in  2             // send a flag signal
00:00:01:010  packet  3000000002// send 2nd data packet
00:00:01:010  packet  2.2
00:00:01:010  packet  2.3
00:00:01:010  packet  2.4
00:00:01:050  outInterface  0   // table's respond, address not found
00:00:02:001  in  3                 // send a flag signal
00:00:03:010  packet  2000000005    // send an update packet
00:00:03:010  packet  3.2
00:00:03:010  packet  0
00:00:03:010  packet  0
00:00:03:050  cont  0               // confirmation from ripTable
00:00:04:001  in  4                 // send a flag signal
00:00:04:010  packet  1000000000    // send a request signal
00:00:04:010  packet  4.2
00:00:04:010  packet  0
00:00:04:010  packet  0
00:00:04:050  cont  0               // confirmation from ripTable
```

**Figure 62: PacketProcessor event file**

```
00:00:00:001  getpacket  1        //requesting a packet from interface 1
00:00:00:030  destination  1.3    //requesting output interface for destination
00:00:00:070  out  3e+06          //forwarding packet through interface
00:00:00:070  out  1.2
00:00:00:070  out  1.3
00:00:00:070  out  1.4
00:00:00:070  out  2
00:00:00:070  next  0            //request next flag
00:00:01:001  getpacket  2       //requesting a packet from interface 2
00:00:01:030  destination  2.3   //requesting output interface for destination
00:00:01:070  out  1e+06
00:00:01:070  out  2.3
00:00:01:070  out  0
00:00:01:070  out  0
00:00:01:070  out  -2
00:00:01:070  next  0             // request next flag
00:00:02:001  getpacket  3        //requesting a packet from interface 3
00:00:03:030  updatetable  3.2    // sending update information to table (address)
00:00:03:030  updatetable  5      // (metric)
00:00:03:030  updatetable  3      // (interface)
00:00:03:050  next  0             // request next flag
00:00:04:001  getpacket  4        //requesting a packet from interface 4
00:00:04:030  requ  4             //forward request info to table (interface)
00:00:04:030  requ  4.2           // (address)
00:00:04:050  next  0             // request next flag
```

**Figure 63: PacketProcessor output file**

The messages in the event file simulated the process of sending flag signals from the queue to packetProcessor, and then responding to the processors requests for packets. We saw from the output file that the model did respond to the three packet types in the correct manner. The model requested the output interface every time it received a data packet -as in the first two packets sent by the event file-. The model used the interface ID to forward the packet, and issued a request to be updated when the interface value received was 0. Using the event file we also simulated an update packet (the 3rd packet) and a request packet (the 4th packet), and for both types the processor outputted the right messages to the ripTable model.

**Testing the RIPTable:**

The RIP table was model in a top level model and messages were injected into the model from an event model RIPTable.ev shown below.

```
00:00:00:010 update 1.1          //sending update data
00:00:00:010 update 1            //metric 1
00:00:00:010 update 5            //interface 5
00:00:00:011 update 1.2          //sending update data
00:00:00:011 update 2            //metric 2
00:00:00:011 update 6            //interface 6
00:00:00:012 update 1.3          //sending update data
00:00:00:012 update 3            //metric 3
00:00:00:012 update 7            //interface 7
00:00:00:013 update 1.4          //sending update data
00:00:00:013 update 4            //metric 4
00:00:00:013 update 8            //interface 8
00:00:00:100 address 1.3         //requesting interface for address 1.3
00:00:00:110 address 1.5         //requesting interface for address 1.5
00:00:00:120 request 1           // request data, address 0 (all table)
00:00:00:120 request 0
00:00:01:010 update 1.3          //update data (address 1.3)
00:00:01:010 update 1            //metric 1
00:00:01:010 update 3            //interface 3
00:00:10:000 request 1           //request update address 0 (all table)
00:00:10:000 request 0
00:00:11:000 request 1           //request update on address 1.2
00:00:11:000 request 1.2
```

**Figure 64: RIPTable event file**

The event file injected events that represent update, request, and destination values. The file started by sending four update value sets, simulating the values that will be passed from the routerProcessor model to the RIPTable. To test that the table did in fact receive the messages in the right format and stored them in its table, two addresses were passed to the model. The first address value was for an address that was passed to the table in one of the update messages, and to this address the model did output the right interface number that was associated with that address. The second address for an address that does not exist in the table, and as expected from the model, the model send the value 0 for the output interface.

To test that the behavior of the model upon receiving a request for all of its table's entries, we send a request with the address filed having the value 0 in it. The model (as shown in the output file –figure 2-) sent all of its routing table entries to that interface port, and followed it with a done signal to the router processor.

The model also accepted updates for an existing address in its table, and did in fact replace the output interface associated with that address, this behavior is seen in the outputted messaged that came as a respond on the second request for the table entries.

```
00:00:00:101 out_interface 7       //out interface 7
00:00:00:111 out_interface 0       //out interface 0 (unknown)
00:00:00:121 out 2e+09             //start of response messages. 1st entry (option)
00:00:00:121 out 1.1               //(address)
00:00:00:121 out 0
00:00:00:121 out 0
00:00:00:121 out 1                 // (interface to respond through)
00:00:00:121 out 2e+09             // 2nd table entry (option filed)
00:00:00:121 out 1.2               //(address)
00:00:00:121 out 0
00:00:00:121 out 0
00:00:00:121 out 1
00:00:00:121 out 2e+09
00:00:00:121 out 1.3
00:00:00:121 out 0
00:00:00:121 out 0
00:00:00:121 out 1
00:00:00:121 out 2e+09
00:00:00:121 out 1.4
00:00:00:121 out 0
00:00:00:121 out 0
```

```
00:00:00:121 out 1
00:00:00:121 done 0                  //responding completed
00:00:10:001 out 2e+09              //start of response messages
00:00:10:001 out 1.1
00:00:10:001 out 0
00:00:10:001 out 0
00:00:10:001 out 1
00:00:10:001 out 2e+09
00:00:10:001 out 1.2
00:00:10:001 out 0
00:00:10:001 out 0
00:00:10:001 out 1
00:00:10:001 out 2e+09
00:00:10:001 out 1.4
00:00:10:001 out 0
00:00:10:001 out 0
00:00:10:001 out 1
00:00:10:001 out 2e+09
00:00:10:001 out 1.3
00:00:10:001 out 0
00:00:10:001 out 0
00:00:10:001 out 1
00:00:10:001 done 0                    //responding completed
00:00:11:001 out 2e+09                //respond with 1 message.
00:00:11:001 out 1.2
00:00:11:001 out 0
00:00:11:001 out 0
00:00:11:001 out 1
00:00:11:001 done 0                    //responding completed
```

**Figure 65: RIP Table output file**

### 4.2.2 Router coupled model test

After successfully testing all the router's components. We used them to created the router's coupled model (the model file is shown in appendix C and tested it using the following event file.

```
00:00:00:010 in1 2000001                        // update with metric 1
00:00:00:010 in1 111101101                     // address
00:00:00:010 in1 0
00:00:00:010 in1 0
00:00:00:020 in1 2000002                        // update with metric 2
00:00:00:020 in1 122202202
00:00:00:020 in1 0
00:00:00:020 in1 0
00:00:00:011 in2 2000003                        // update with metric 3
00:00:00:011 in2 133303303
00:00:00:011 in2 0
```

```
00:00:00:011 in2 0
00:00:00:030 in2 2000004                    // update with metric 4
00:00:00:030 in2 145525056
00:00:00:030 in2 0
00:00:00:030 in2 0
00:00:00:040 in1 2000005                    // update with metric 5
00:00:00:040 in1 115001151
00:00:00:040 in1 0
00:00:00:040 in1 0
00:00:00:100 in1 3010012                    // data, ttl=10, CRC=12
00:00:00:100 in1 121117001                  // source address
00:00:00:100 in1 133303303                  // destination address
00:00:00:100 in1 15
00:00:01:010 in1 2000000                    // update metric 0
00:00:01:010 in1 133303303
00:00:01:010 in1 0
00:00:01:010 in1 0
00:00:01:220 in1 1000000                    // request
00:00:01:220 in1 0
00:00:01:220 in1 0
00:00:01:220 in1 0
00:00:02:000 in1 3008011                    // data, ttl = 8, CRC = 11
00:00:02:000 in1 114124201
00:00:02:000 in1 123456789                  // unknown destination
00:00:02:000 in1 0
00:00:02:010 in2 2000007                    // update metric 7
00:00:02:010 in2 122202202
00:00:02:010 in2 0
00:00:02:010 in2 0
00:00:02:010 in1 3000007                    // data, TTL = 0
00:00:02:010 in1 122202202
00:00:02:010 in1 0
00:00:02:010 in1 0
```

**Figure 66: Router event file**

The desire was to test the router for all possible types of traffic expected. Going through the event file will show that the behavior of the router was as expected.

The first 5 packets were update packets. The router did pass the related values to its table and the table updated it self as tested before. We can see that for every update packet, an update the neighbor nodes was sent thought the other router interface. For example taking the 1st update message, we can see it arrived at the router from interface 1, and that a corresponding update message was created and sent through interface 2.

After the update messages, a packet representing a data packet was injected into the router. The packet option files shows a TTL value of 10. The router knew the address

since it received an update on it before (this shown in the blue highlighted sections in the event file). The router did forward the packet using the right output interface as shown in the output file below.

After this packet, another update with a smaller metric for an address that the router has in its table was sent through interface 1. We can see in the output file that the router did update its table with the better metric value and sent an update through interface 2 (highlighted in yellow in both event and output file).

A request was sent at time (00:00:01:220) requesting the full routing table, and as seen in the output file, the router did send all of its table entries to the requesting nod.

No output was sent in response to the last two packets. The reason is that the first one was an update with a metric higher than the existing one in the routing table. The second was a data packet with a TTL value of 0 (expired). In both cases the router discarded the packets.

```
00:00:00:018 out2 2e+06                    // update
00:00:00:018 out2 1.11101e+08               // address
00:00:00:018 out2 0
00:00:00:018 out2 0
00:00:00:023 out1 2e+06                     // update
00:00:00:023 out1 1.33303e+08
00:00:00:023 out1 0
00:00:00:023 out1 0
00:00:00:028 out2 2e+06                     // update
00:00:00:028 out2 1.22202e+08
00:00:00:028 out2 0
00:00:00:028 out2 0
00:00:00:038 out1 2e+06                     // update
00:00:00:038 out1 1.45525e+08
00:00:00:038 out1 0
00:00:00:038 out1 0
00:00:00:048 out2 2.00001e+06              // update
00:00:00:048 out2 1.15001e+08
00:00:00:048 out2 0
00:00:00:048 out2 0
00:00:00:109 out2 3.00901e+06              // data forward
00:00:00:109 out2 1.21117e+08
00:00:00:109 out2 1.33303e+08
00:00:00:109 out2 15
```

```
00:00:01:018 out2 2e+06                    // updtae
00:00:01:018 out2 1.33303e+08
00:00:01:018 out2 0
00:00:01:018 out2 0
00:00:01:228 out1 2e+09                     // respond
00:00:01:228 out1 1.11101e+08
00:00:01:228 out1 0
00:00:01:228 out1 0
00:00:01:229 out1 2e+09                     // respond
00:00:01:229 out1 1.22202e+08
00:00:01:229 out1 0
00:00:01:229 out1 0
00:00:01:230 out1 2e+09                     // respond
00:00:01:230 out1 1.45525e+08
00:00:01:230 out1 0
00:00:01:230 out1 0
00:00:01:231 out1 2e+09                     // respond
00:00:01:231 out1 1.15001e+08
00:00:01:231 out1 0
00:00:01:231 out1 0
00:00:01:232 out1 2e+09                     // respond
00:00:01:232 out1 1.33303e+08
00:00:01:232 out1 0
00:00:01:232 out1 0
00:00:02:009 out2 1e+06                     // reauest
00:00:02:009 out2 1.23457e+08
00:00:02:009 out2 0
00:00:02:009 out2 0
```

**Figure 67: Router output file**

# 5 Conclusion

In conclusion, the project at hand is *building a library for parallel simulation of networking protocols*. The main protocol of interest is TCP/IP for its wide use and variety of applications and magnitude of services provided. The project was successful in creating a library of models capable of building simple topologies as a first step for building a more complicated library in future projects. The project has surveyed current network simulation tools available and deduced an advantage for a network simulator based on the DEVS formalism. Some of the models where successfully imported to a parallel environment, thus allowing for a parallel simulator to be built on top of the model library. Further more; we have managed to survey prominent researchers in the field of network and parallel simulation. This helped in the decision process when choosing the library models.

The models chosen are sufficient to create simple network topologies with an acceptable level of accuracy in services, and customization in terms of Quality of service parameters, and Service level agreements. The models created provide the backbone for a larger model library building on top of it, since all components chosen, represented different fields and layers of a typical packet switched network.

# 6 Recommendations

Recommendations for future work are generally based on problems the project faced, preventing us from pursuing the development of a DEVS network simulator in a parallel environment.

A major problem facing us, is the Model file size. Size of the files started to get larger and larger as models got more complicated, an example of that is the host-coupled model. attempts to use the “Macro” function[ 17], proved not to work, since it only lets the user have the tool copy a chunk of copy that doesn't change from one file to another, although very useful in making model files more readable, but not useful when attempting to create instances of let's say the host model file. For that we recommend the development of a model file maker according to the specifications of the models in the library we require to develop a function inside of the tool to create instances of models (I.e: change port names, connection names, and preparation time) to facilitate writing large model files of complex topologies, or developing a compiler based on the GCC compiler, to have models as classes and facilitate instantiating them.

It is also apparent that the library must be extensively expanded to allow for a more user friendly operation of a case tool that would analyze networks, especially elements like ATM switches, DWDM devices [16], DSU/CSU units [16], and other core network units.

Development of such models and enhancing the current models would allow for building of larger topologies, and at a certain point an image of the Internet, since the Internet is large network of smaller networks. Library expansion should also branch into building models for devices such as Voice over IP (VOIP) devices such as phones, for the large popularity they are gaining, and to analyze their effect on current network structure in terms of load, and to access their credibility.

Another recommendation, in order to complete a parallel simulator is of course the Conversion of current library models to parallel, as well as building new models in the parallel environment, is a must to accomplish the bigger picture of the project, we have managed to step into the parallel paradigm, by means of converting some of our models into parallel, the research presented in Appendix “B” should shed some light on our efforts in this field, and provide a starting point for future work.

Although not an immediate requirement to build a simulator, however a topology reader, meaning a software package capable of reading specific file formats, NS-2[8] topologies for an example and converting them into model files based on the library models, so as to facilitate the analysis process of currently present networks, being studied with some of the currently present tools.

# 7 References

[1] VINT project
[2] (Zeigler 1976, Zeigler at al.2000)
[3] CD++ toolkit
[4] www.red.net/glossary/a.php
[5] 4602 notes G&H chapter 8.4, 8.5
[6] RFC editor website
[7] William Stallings, *Data and Computer Communications.* Upper Saddle River, New Jersey: Prentice-Hall, 2000.

[8] Gorry Fairhurst, "Carrier Sense Multiple Access with Collision Detection (CSMA/CD)," [Online Document], January 1st 2004 [March 20, 2004]. Available: http://www.erg.abdn.ac.uk/users/gorry/course/lan-pages/csma-cd.html

[9] [Online document], Available: http://www.linux-praxis.de/lpi101/p-glossary.html

[10] [Online document], Available: http://www.mminternet.com/dsl/glossary.htm

[11] [Online document], Available: http://www.rvcomp.com/EIA/glossary.htm

[12] [Online document], Available: http://www.netbenefit.com/support_glossary.html

[13] G. Malkin, "RIP Version 2", *Network Working Group*, Request for Comments: 2453, November 1998

[14] C. Hedrick, "Routing information Protocol", *Network Working Group*, Request for Comments: 1058, June 1988

[15] [Online document], Available: http://www.stallion.com/html/support/glossary.html

[16] [Online document], Available: http://www.pace.ch.cours/glossary.htm

[17] [Online document], available: http://www.freesoft.org/CIE/Course/Section4/8.htm

[18] PCD++ manual

# 8.0 Appendix List