Design and Implementation of a Library of Network Protocols in CD++

Mohamed Abd El-Salam Ahmed Khalil Yonis
Abdul-Rahman Elshafei Gabriel Wainer
Department of Systems and Computer Engineering
Carleton University. 1125 Colonel By Dr. Ottawa, ON. K1S 5B6. Canada
{maesalam, kyonis, abdul}@connect.carleton.ca
gwainer@sce.carleton.ca

Abstract

Simulation-based analysis can help to design, study and configure computer networks, in order to assess the best possible solution to particular problems. We propose the creation of a tool for modeling and simulation (M&S) of networks built on the DEVS formalism. DEVS allows for the formal definition of discrete event models interacting together, which can be used to analyze properties about the systems we model. These models can be executed under different simulation environments, and they can be easily integrated with models of other phenomena, described with different techniques. Here, we present the definition and implementation of a library of DEVS models constructed as the initial building block for such a network simulator based on the CD++ tool.

1. Introduction

Scale and heterogeneity brings two major sources of stress to the design of protocol in networking technologies. Scale affects both the correctness and performance of a network, while heterogeneity of applications translates into a large number of interacting protocols and traffic patterns, with varied requirements [1].

In order to produce robust and evolvable networking technologies, we need to analyze the dynamic behavior of the underlying protocols. The complexity of this task has made M&S a widely used technique for addressing a variety of problems in this field. Although various discrete-event simulators are readily available (both academic and commercial such as NS-2 [2], OPNET [3], and OMNet++ [4]), a new M&S tool based on the DEVS formalism [5] could add different advantages. DEVS provides a formal foundation to M&S that proved to be successful in different complex systems. DEVS combines the advantages of a simulation-based approach with the rigor of a formal methodology. The formalism is based on sound theoretical grounds, allowing for an abstract design of models that are independent from the implementation platform and running conditions. The creation of a DEVS-based network simulator could provide:

- Facilities to carry out formal tests.
- Seamless model sharing between different DEVS-based toolkits [6].
- High-performance execution of the same models in a parallel simulation environment [7].
- Remote execution using client-server services, allowing remote interaction between users [8].
- The ability to execute the models on a distributed platform based on the HLA, Corba or other technologies [9],[10],[11].
- The possibility to define models using different techniques interacting within the same environment [12]. This could allow including non-network entities that affect network operation, providing results that are more realistic.
- The potential to automatically deploy models that have been tested on the simulation environment into the actual networking hardware, converting them into the real application [13], [14].

We will discuss the design and implementation of such a simulator as a library for CD++ [6], a DEVS-based toolkit. The library is able to simulate user-defined topologies to assess network functionality; modular design allows the addition of new models easily, while the models themselves are flexible to permit future enhancements. We will present the general ideas about the design and discuss different issues about the implementation of this library.

2. Background

DEVS (**D**iscrete-**Ev**ent system **S**pecifications) is a systems theoretic approach to M&S, which allows the definition of hierarchical modular models [5]. A system modeled with DEVS is described as a composite of **atomic** or **coupled** sub-models. DEVS atomic models are described as:

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

where ***X*** is a set of input events, **S** is a set of discrete states, ***Y*** is a set of output events, $d_{\mathbf{int}}$ is an internal transition function, $d_{\mathbf{ext}}$ is an external transition function, l is an output function, and **ta** is a lifetime function. Every state has a lifetime. When this time is consumed, the model activates l (providing outputs), and changes to a new state determined by $d_{\mathbf{int}}$. If a model receives an input event, $d_{\mathbf{ext}}$ is triggered. This function uses the input value, the current state and the time elapsed since the last event in order to determine which is the next model's state.

A DEVS coupled model is defined as:

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

where ***X*** is a set of input events, ***Y*** is a set of output events, **D** is an index of components, and for each i ∈ D, $\mathbf{M_i}$ is a basic DEVS model (atomic or coupled), and $\mathbf{I_i}$ is a set of influencees of model i (which determines the destination models for the outputs). For each j ∈ I_i, $\mathbf{Z_{ij}}$ is an i to j translation function, which converts the outputs of a model into inputs for its influencees.

CD++ [6] is an M&S toolkit based on DEVS specifications. Atomic models are programmed in C++, and coupled models are defined with a built-in specification language. CD++ is built as a class hierarchy of models related with simulation processing entities. DEVS Atomic models can be programmed and incorporated onto a *Model* base class hierarchy in C++. A new atomic model is created as a new class that inherits from the *Atomic* base class. The state of a model is defined in the *AtomicState* class.

```
class Atomic : public Model  {
public:
virtual ~Atomic();   // Destructor

protected:
//Kernel services
Time nextChange();
Time lastChange();
holdIn(AtomicState::State &, Time &);
passivate();
ModelState* getCurrentState() ;
sendOutput(Time   &time,Port   &port,Value
value);
```

```
//User defined functions.
initFunction();
externalFunction(ExternalMessage & );
internalFunction(InternalMessage & );
outputFunction(CollectMessage & );
string className() const
};     // class Atomic
```

Figure 1. The Atomic class in CD++

The *Atomic* abstract class defines some service functions: **`nextChange/lastChange`** return the time until/from the next/last event; **`holdIn`** defines DEVS *ta* function; **`passivate`** sets the next internal transition time to infinity (the model will only be activated again if an external event is received); **`getCurrentState`** returns the current model's phase; **`sendOutput`** sends an output message through the specified *port*.

A newly defined atomic model should override the following methods: **`initFunction`**, invoked at the first activation of the model; **`externalFunction`**, the δ_{ext} function of the DEVS; **`internalFunction`**, which defines the δ_{int} function; and **`outputFunction`**, the DEVS λ function.

Once an atomic model is defined, it can be combined with others into a coupled model, as follows:

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

[router_out]
preparation : 050
```

Figure 2. RouterOut coupled model

This figure shows how to define one of the components of the Router model, introduced later in section 3.2. The topmost coupled component is called *top*. After a model's name is defined, a list of sub-components (either an instance of an atomic model or another component) is defined using the *components* keyword. Then, a list of input and output ports is defined for the model, using the keywords *in* and *out* respectively. Once the models' ports are defined, their coupling can be described using the *link* keyword, followed by the output port for the event, and the input port that will receive it.

3. Library Components

The simulator has been built as a library of models in CD++, and it consists of two major units: data generators and inter-networking devices. Data generators are modeled with a model (named *host*), which is based on emulation of the TCP/IP protocol stack. Inter-networking devices mo dels include a *router* and a *hub*, which gives the library the initial depth needed to simulate complex topologies. The models are built in a modular fashion to maintain a high degree of flexibility and customizability for future development of the tool. Furthermore, the mo dels 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. We started by formally specifying each model, and by studying its basic behavior through the formal specifications. In the following subsections, we will include a brief explanation about each component. Detailed model specification and implementation details can be found in [15].

3.1. Host

The Host coupled model simulates the layers of the TCP/IP protocol stack, and it is thus comprised of distinct models representing the Application, Transport, Network, Data Link, and physical layers. The structure of the coupled model is shown in Figure 3. Host Coupled Model.

We will start by discussing the definition of the Transport layer, which contains some of the most complex models. As we can see in Figure 3, the TCP model was broken in two (to facilitate full-duplex communications). The Transmitter module is responsible for receiving data from the Application layer model, adding sequence and acknowledgment numbers, a window size and a checksum to the original data received (fields required for Service Level Agreement simulations).

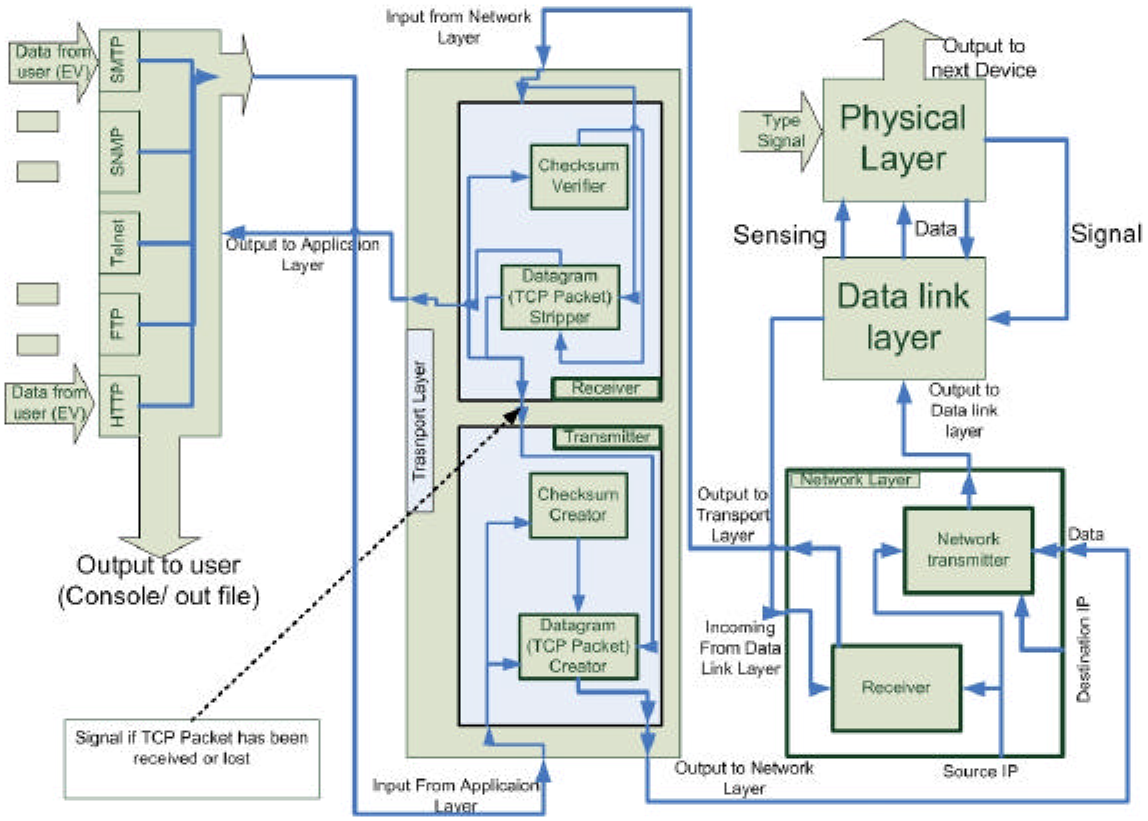

Figure 3. Host Coupled Model

The data received is transformed to the format shown in Figure 4, conforming the protocol requirements [16].

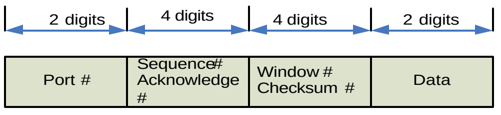

Figure 4. TCP Packet format

Packet creation is split between two atomic models; *datagramCreator* and *checksumCreator*. Data received (from the Application layer) is routed to the *datagram-Creator*, which will create an initial packet and forward it to the *checksumCreator* to compute a checksum. Then, the completed packet will be forwarded to the *datagramCreator*, and sent to the next layer in the protocol stack. Before the packet is sent, a copy is saved to accommodate the connection manager, which will resend packets in case they are not received. Each of the models was formally specified, as follows:

datagramCreator = < X, S, Y, d_{int}, d_{ext}, λ, D>

X = { *in*: receives data; *Checkin*: receives packets after the checksum has been created; *ackPort*: receives acknowledgments; *ackSender*: receives requests to send ACKs (the data received) };

S = { phase, packet, saved packet, delay}

Y = { *gocheck*: sends data packets to request the creation of a checksum; *datagramCreator*: sends complete output data packets; *resend* used to retransmit packets };

```
$\delta_{int}$(s,e) :
Case phase
        active: passivate;

$\delta_{ext}$(s,e,x) :
  Case msg.port
        In: Create packet;
        Checkin: packet received, checksum added
        ackPort: check acknowledgement
                correct?: delete saved packet
                else: resend saved packet
        ackSender: send received data as ACK.
        phase = active; holdIn(delay);

$\lambda$(s) :
If message = packet and no checksum yet
  Send packet through gocheck (checksum = 0)
If message = data and checksum created
  Send data on datagramCreator
If message = ack
        Check ACK to be correct or not.
        Incorrect? Discard ack; resend packet.
If message = request to send ack
  Send message on resend.
```

On the receiver side, we created a model to receive data from the Network layer. The model is made of two atomic components: a *datagramStripper* and a *checksumValidator*. The *datagramStripper* receives data and forwards it to the *checksumValidator* to test the checksum. If valid, the *datagramStripper* checks the packet type (data or ack). If it is data, the headers are stripped, the data is forwarded, and a request to the *datagramCreator* to send an ack to the source of the packet is issued. On the other hand, if the data is an ack, the *datagramStripper* forwards it to the *datagramCreator* to check if the ack is expected (either deleting the saved packet or resending it). If the checksum is incorrect, the packet is discarded.

After careful study of the model's specifications, every model was coded using the CD++ services presented in Figure 1. Models were individually tested, and coupled models were created using the notation presented in Figure 2. Finally, integration testing was carried out. Figure 5 shows a detailed execution log of the Transport Layer model, in which we present the data being manipulated by its various models (more information on the internal structure and design of the layer can be found in [15]).

```
X/00:10:000/top/in/1280 to datagramcreator
D/00:10:000/datagramcreator/005 to top
*/00:10:005/top to datagramcreator
Y/00:10:005/datagramcreator/gocheck/
             1200000000080 to top
D/00:10:005/datagramcreator/... to top
X/00:10:005/top/in/1200000000080
            to checksumcreator
D/00:10:005/checksumcreator/005 to top
*/00:10:010/top to checksumcreator
Y/00:10:010/checksumcreator/checksum-
creatorout/1200000009280 to top
D/00:10:010/checksumcreator/... to top
X/00:10:010/top/checkin/1200000009280
              to datagramcreator
D/00:10:010/datagramcreator/005 to top
*/00:10:015/top to datagramcreator
Y/00:10:015/datagramcreator/datagram-
creatorout/1200000009280 to top
...
```

Figure 5. Transport layer log file

The first event (X) is an input carrying the value 12 through the HTTP port 80. This is transmitted to *datagramcreator*, which executes the external transition function (adding the window size, acknowledgment, and sequence number to the data - for testing purposes, the values = 0). Then, it schedules an internal transition (D) in 5 ms (reflecting the delay of the circuit). When this time is consumed, an internal transition (*) is fired. The first step involves executing the output function (Y), which transmits the packet through the *gocheck* port. The model then passivates ("…" represents time=∞). This event is converted into an input (X) for *checksumcreator*, which receives the Application data and computes the checksum (also taking 5 ms). Once the checksum is computed, it is sent to the *datagramcreator* to signal that it is ready to be sent.

The data presented on the previous example arrived to the Transport layer through the higher level Application layer. This layer models a host generating and receiving data routed from various inter-networking devices, and different services and protocols. The data generated is depicted in Figure 6. The Application layer receiving this data adds Application port variable, and then to outputs the information to the Transport layer (for instance, the HTTP request generated at 09:500 is the one originating the sequence presented in Figure 5).

```
INPUT
// data input on HTTP input port
00:07:000 infromHTTPuser 11
00:09:500 infromHTTPuser 12
...
// data input on Port 25
03:00:00 infromSMTPuser 13
03:10:00 infromSMTPuser 14

OUTPUT
//Application data sent on HTTP Port
00:07:500 outtoTransport 1180
00:10:000 outtoTransport 1280
...
// Application data sent on Port 25
03:00:480 outtoTransport 1325
03:10:530 outtoTransport 1425
```

Figure 6. Application output file

The Network layer is usually where most of the delay and stochastic operation occurs due to the nature of IP being a connectionless protocol. The layer adds a source and destination IP fields to the packet to enable routing, creating subnets, local networks, and many other Network artifacts, as showed in the following figure.

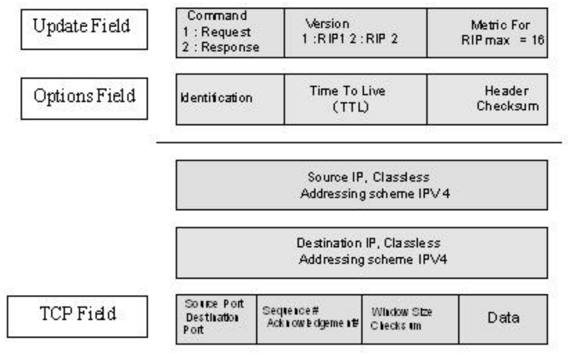

Figure 7: Header Format

The headers for the Internet Protocol are based on RFC # 791 [16]. 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, and a checksum. The traffic packets are made of four values: the source address, the destination address, and the TCP field. The options in each field are chosen from the IPV4 packet format. As seen in Figure 3, the Network model consists of a transmitter and a receiver, which add/extract the corresponding information using the header format in Figure 7. Figure 8 shows an input example for this model.

```
010 infromTransport 1122334455580  // data
020 DestinationIP 192168111223     // IP
value
```

Figure 8. IP test values

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, as follows:

```
Y/00:13:020/netwXmit1/out/4850000155000 to
top
Y/00:13:020/netwXmit1/out/1921681162240 to
top
Y/00:13:020/netwXmit1/out/1921681162240 to
top
Y/00:13:020/netwXmit1/out/122233343180800
to top
```

Figure 9. Network layer log file

The Data Link layer in our model implements the CRC operations of the Logical Link Control (LLC) sub layer (which calculates a frame, checks the sequence, and uses it to detect errors when a frame is received), and the Carrier Sense Multiple Access with collision detection (CSMA) algorithm in the Medium Access Control (MAC) sub layer, which would senses the carrier by sending a *senseCarrier* port message to the Physical layer, and waits for a response.

The Physical layer model simulates the wiring connecting different devices, using a list to save the incoming data, and outputting them at a specific time intervals, modeling the link delays. As seen in Figure 3, the model interacts with others through the Data link Layer and the *sensing* port. The layer can have one of four states (*idle*, *busy*, *jammed*, *collision*) that determine how data is handled.

Data is received seamlessly through the same set of layers in the reverse direction with each layer stripping the extra variables added by its counterpart. The following figures show the results of one of the integration tests for a host whose source IP address is 111222333.

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

Figure 10. Integration test suite

The event file shows FTP data from the host to another end on the network. Simple values where chosen here, to ease the process of reviewing the results. The host reacted to these events, as shown in the following figure.

```
Y/49:010/netwXmit1/out/2000000000 to top
Y/49:010/netwXmit1/out/111222333 to top
D/49:010/netwXmit1/... to top
Y/49:010/top/outtoData Link/2000000000 to
Root
Y/49:010/top/outtoData Link/111222333 to
Root
```

Figure 11. Host log file section

This section of the host log file shows two events, the first represents the host sending the received data, throughout the network (after adding the appropriate headers), forwarding it to the and that the Data Link layer. The Data Link actually responded as in the following figure

```
Y/06:000/internet/outtoData Link/ 20000 to
top
Y/06:000/internet/outtoData
Link/192168116224 to top
D/06:000/internet/... to top
X/06:000/top/getpacket/ 20000 to Data Link
X/06:000/top/getpacket/192168116224 to
Data Link
```

Figure 12. Data link interaction.

Figure 12 shows the output of data from the Network layer to the Data Link 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.

3.3 Hub

Hubs are simple layer one devices that simply regenerate received data to all connected devices. The decision to include a model of a Hub into the initial library design was to make use of its simplicity to experiment. The coupled model design is as follows

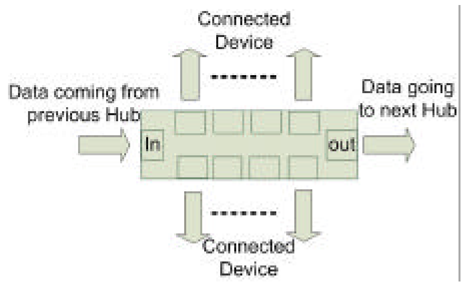

Figure 13. Hub structure

The Hub Atomic Model specification is:

$$Hub = < X, S, Y, d_{int}, d_{ext}, \lambda, D>$$

X = { *In*: receives data from interconnected devices; *Set*: sets hub specific information };

S = {Sigma, X, Preparation Time}

Y = { *Out1..n*: 1st…nth connected device };

```
d_int (e, s): {
        case phase:
                active: passivate
}
d_ext(s, e, x): {
        case msg.port:
          In: set localvalue to msg.value
          Set: set local data field (hub identifier)
                to msg.value
}
```

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

Any connected device can send data onto the hub to be broadcasted to all other devices, however one must pay close attention to the timing at which events are sent otherwise data might be lost. The model proved successful in linking multiple Hosts together providing simple local networks and proved useful in creating subnets.

3.2 Router

The router model defines how to interconnect network devices. We used an abstract look in the routing process, considering three main functionalities: receiving and forwarding traffic, processing IP packets, and maintaining a routing table.

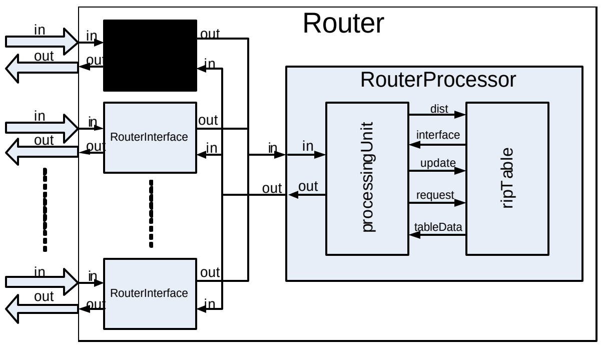

Figure 14. Router's coupled module

In order to simulate the three functions, two models were created; the *RouterInterface* and the *RouterProcessor*, which in turn is made of the *ProcessingUnit* model and a *ripTable*. The router coupled model is shown in Figure 14 (each of the models is even subdivided in lower levels of abstraction that are not discussed here). Every router has a number of interfacing cards receive/forward traffic from/to the network. The *RouterInterface* model was developed to receive and send packets with the format discussed in Figure 7. To handle the traffic going in/out of the router, the *Router-Interface* was designed as a coupled model consisting of one model receiving packets from the network, and a second one forwarding packets out of the router. After packets are received by the *RouterInterface*, they are processed to see if they are messages to the router (requests or updates), or just data packets to be forwarded to their destinations.

The *ProcessingUnit* is responsible for reading in the packets from the interfaces, processing them, and making routing decisions regarding their destinations. Upon receiving a packet, it looks at the packet's header, extracts from it the TTL value, and checks if it is valid. In that case, it will read the packet's type, and it will react according to type.

Three types of packets are accepted: *respond*, *request*, and *data*. *Request* packets carry the destination address of the requesting router that wants the update, following the RIP protocol [17]. This address value is extracted from the packet's header and it is sent along with the requesting router reply information to the *ripTable* model, so the proper reply information can be prepared and sent to the requesting router. The *respond* packets carry a network address and a metric value (cost) associated with that route. These packets are used to update other routers, or to respond to other routers' requests for updates. The router extracts both the address and the metric, and it forwards this information along with the sending router's data to the *ripTable*. When a *data* packet is received, the *processing Unit* extracts its destination address, and forwards it to the *ripTable* model (which maintains the routing information for forwarding packets). Once the *ripTable* returns an output interface, the *RouterProcessor* will simply forward the data packet through it. If the destination address is not found in the routing table, the value 0 is returned (and a request packet is issued through all interfaces except the one that the packet was received through, requesting an update on that destination).

The *ripTable* is in charge of maintaining the routing information that the router needs to forward packets to its destinations. The entries in the table have the format <*Address*, *Metric*, *Interface*>. *Address* is a destination for the packet; *Metric* represents the cost of getting to that destination, and the output *Interface* is the one through which the router must forward the packet (in order to be at least one hop closer to the destination).

The *ripTable* receives three events: *update*, *request*, and *request for forwarding* information. In the case of *updates*, the model will be receiving the address that the update is about, together with a new metric value. If the address does not exist, the information will be added. Otherwise, the associated metric is compared with the newly received one, and it will replace the output interface number with that of the new update, if the new metric value is smaller than the one in the table. For the *request* events, the *ripTable* model will prepare the required information from its table, and redirects it as responds. Finally, for the *forward information* request, the model will search its table for the destination address and send the output interface that should be used to forward the packet.

The router's behavior was tested using different scenarios, as showed in Figure 15.

```
INPUTS
00:00:010 in1 2000001  // update with met-
ric 1
00:00:010 in1 111101101  // address
...
00:00:100 in1 3010012 // data, ttl=10,
CRC=12
00:00:100 in1 121117001 // source address
00:00:100 in1 133303303 // destination ad-
dress
00:00:100 in1 15
00:01:010 in1 2000000   // update metric 0
00:01:010 in1 133303303
...
00:02:000 in1 3008011 // data,ttl=8, CRC =
11
00:02:000 in1 114124201
00:02:000 in1 123456789 // unknown desti-
nation 00:02:010 in2 2000007   // update
metric 7
00:02:010 in2 122202202
00:02:010 in1 3000007  // data, TTL = 0
00:02:010 in1 122202202

OUTPUTS
00:00:018 out2 2000001  // update
00:00:018 out2 111101101  // address
...
00:00:109 out2 3010012 // data forward
00:00:109 out2 121117001
00:00:109 out2 133303303
00:00:109 out2 15
00:01:018 out2 2000000   // update
00:01:018 out2 133303303
...
00:02:009 out2 1000000// request
00:02:009 out2 123456789
```

Figure 15. Router Input/Output events

The first packet is an update. The router passes the related values to its table and the table is updated. The message arrived at the router from interface 1, and a corresponding update message was created and sent through interface 2 (*out2*). For every update packet, an update to the neighbor nodes is sent thought the other router interface. Then, we show a packet representing data injected into the router. The packet option field shows a TTL value of 10. The router knew the address since it received an update on it before. The router forwards the packet using the right output interface. After, another update with a smaller metric for an address that the router has in its table is sent through interface 1. We can see that the router did update its table with the better metric value and sent an update through interface 2.

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.

4. Conclusion

We have showed how to define models of internetworking devices such as routers and hubs, and studied the operation of TCP/IP using a DEVS-based approach. We created a library of models capable of building topologies as a first step for building a more complex library.

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

A DEVS Network simulator advantages from the point of view of scalability and ease of use in terms of creating and adding new components to the library. The selected set of protocols and devices can be used as a template, since they cover all aspects of packet switched networks with enough details to allow for customization and variation.

Acknowledgments

This work has been partially supported by NSERC (National Science and Engineering Research Council of Canada) and the Intel IXA University program.

References

[1] C. Hedrick. “Routing Information Protocol,” *Network Working Group*, Request for Comments: 1058, June 1988.

[2] “The Network Simulator-NS-2”. http://www.isi.edu/nsnam/ns/, last visited: 28/11/03.

[3] X. Chang. “Network Simulations with OPNET”. *Proceedings of the 31st Winter Simulation Conference*. Phoenix, AZ. 1999.

[4] A. Varga. “The OMNeT++ Discrete Event Simulation System”. *Proceedings of the European Simulation Multiconference*. Prague, Czech Republic. 2001.

[5] B. Zeigler, T. Kim, and H. Praehofer. *Theory of Modeling and Simulation*. Academic Press. 2000.

[6] G. Wainer. “CD++: a toolkit to develop DEVS models”. *Software-Practice and Exp*. 32, 1261-1306. 2002.

[7] A. Troccoli and G. Wainer. “Implementing Parallel Cell-DEVS”. *Proceedings of Annual Simulation Symposium*. Orlando, FL. U.S.A. 2003.

[8] G. Wainer and W. Chen. "A framework for remote execution and visualization of Cell-DEVS models". *Simulation*. Vol. 79, pp. 626-647. November 2003.

[9] B. Zeigler, H. Cho, J. Lee and H. Sarjoughian. *The DEVS/HLA Distributed Simulation Environment And Its Support for Predictive Filtering*. DARPA Contract N6133997K-0007: ECE Dept., UA, Tucson, AZ. 1998.

[10] C. Zhang *Integrating existing DEVS simulations with the HLA*. M.A.Sc. Thesis. Carleton University. 2004.

[11] Y.W. Cho, X. Hu, and B. Zeigler. “The RTDEVS/CORBA Environment for Simulation-Based Design of Distributed Real-Time Systems”. *Simulation*, Vol. 79, No. 4, 197-210. 2003.

[12] P. MacSween and G. Wainer. "On the Construction of Complex Models Using Reusable Components". In *Proceedings of SISO Spring Interoperability Workshop*. Arlington, VA. U.S.A. 2004.

[13] E. Glinsky and G. Wainer. Modeling and simulation of systems with hardware-in-the-loop". In *Proceedings of the Winter Simulation Conference*. Washington, DC. 2004.

[14] G. Wainer, E. Glinsky and P. MacSween. “Model-Driven Architecture of Real-Time Systems”. Accepted for publication in *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*. S. Beydeda and V. Gruhn eds., Springer-Verlag (expected date of publication: April 2005).

[15] M. Ahmed, A.-R. Elsahfei and K. Yonis. “Building a library for parallel simulation of networking protocols”. [Online]. Dept. of Systems and Computer Engineering. http://www.sce.carleton.ca/faculty/wainer/students/DEVSnet/index.html. Carleton University. 2004.

[16] RFC-editor “Official Internet Protocol Standards”, [Online]. ftp://ftp.rfc-editor.org/in-notes/rfc791.txt. Accessed: [2003 Sep. 24].

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