

# Simulating Virtual Circuits in Mobile Packet Radio Networks

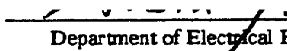
*Alain Jerome Cohen*

Submitted to the Department of  
Electrical Engineering and Computer Science in Partial Ful-  
fillment of the Requirements for the Degree of Bachelor of  
Science in Electrical Engineering Massachusetts Institute of  
Technology, May 1986.

Copyright Alain Jerome Cohen 1986

The author hereby grants to M.I.T. permission to reproduce and to  
distribute copies of this thesis document in whole or in part.

Signature of Author:

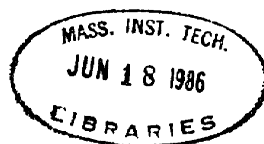
  
Department of Electrical Engineering and Computer Science; May 9, 1986

Certified by:

  
Dimitri P. Bersekas, Thesis Supervisor

Accepted by:

  
David Adler, Department Committee Chairman



Archives

## Table of Contents

1.	Introduction	1
2.	Broadcast Networks	2
3.	OPNET Fundamentals	
3.1.	Background	3
3.2.	Overview	4
3.3.	Structure	7
3.4.	Network Specification	11
3.5.	Device Specification	18
4.	OPNET Enhancements	26
5.	Case Study	
5.1.	Background	28
5.2.	Problem	29
5.3.	Virtual Circuits in MPRNs	29
5.4.	Models	36
5.5.	Simulation Results	37
5.6.	Conclusion	38
6.	References	39
7.	Appendix	40

## **1. Introduction**

Packet radio networks are used in distributed communication systems where nodes are mobile or resources are located in inaccessible areas. In the case where the nodes are mobile, no guarantees can be made about the connectivity between any pair of nodes which wish to initiate a conversation. Moreover the network connectivity may well change during the conversation itself.

The problem of routing in static networks, in which a valid network map is available to all nodes at all times, is to select the best available communication path so as to optimize use of communication resources. In mobile networks, the problem starts at a much more basic level, since the node originating the conversation generally has no way of knowing where the destination node is located. Since node positions are arbitrary, in order to acquire the knowledge of a path to reach the destination node, the origin must rely on a flooding process, in which all the nodes of the network are contacted.

The flooding process may or may not be invoked by the origin node. If the flood is not to start at the origin node, then one must envisage some sort of centralized resource shared by the entire network. This resource would be responsible for maintaining an up-to-date network map, and either informing the ordinary nodes of the proper routes to use, or performing the routing itself in a centralized fashion. On the other hand, if the flood is begun at the origin node, no centralized resources are needed, as each node of the network is responsible for acquiring its own routing information.

The protocols executed by the nodes were developed using OPNET, a simulation package available at the Laboratory for Information and Decision Systems. The goal of this thesis is to extend the class of networks which can be studied with OPNET and to confirm that the new capabilities are adequate through the construction of a detailed, practical model. The model involves three protocol layers supporting communication in packet radio networks with virtual circuit sessions.

## **2. Broadcast Networks**

In this section, some of the differences between broadcast radio networks and point-to-point networks are presented.

- 1) In point-to-point networks a communication channel is associated with its origin and destination nodes. Thus receiving nodes may implicitly know the identity of the transmitting nodes. Therefore Packets do not have to be addressed to their immediate destinations. In radio networks however, channels are shared among many nodes. When a node transmits a packet, it will potentially be received by all nodes within range. In order to transmit to only one destination node, a transmitter must label the packet with the appropriate destination; all nodes will discard packets whose address labels are not equal to the node address.
- 2) When a node is in receipt of several packets simultaneously, these packets are considered void since the receiver in reality sees the superposition of the various signals which generally is meaningless. In some cases, one signal dominates the others and a single packet survives the collision. This phenomenon, known as the capture effect [1], may be due to the difference in distance or power of the multiple transmitters. Throughout this paper the capture effect will be ignored and collided packets considered lost.
- 3) Some schemes for transmission failure control may take advantage of the fact that when a node transmits a packet it will hear it rebroadcast by the receiving nodes shortly thereafter. Thus no explicit acknowledgements are necessary; if a node receives the packet which it has recently transmitted, then it may assume that it was correctly received by the destination node. This technique is particularly applicable to satellite systems in which all nodes are guaranteed to receive the packets rebroadcast by the satellite. Unfortunately, this scheme encounters certain difficulties in multi-hop packet radio networks, as will be seen later.

### **3. OPNET Fundamentals**

*Note:* This section is largely from the *OPNET User Manual* co-authored with Steven Baraniuk. Thus passages may be in common with Steven Baraniuk's S.B thesis [2].

#### **3.1 Background**

OPNET, which stands for Ordinal Process Network Evaluation Tool, is a group of programs designed to help researchers simulate and study distributed protocols, in the context of packet-switched networks. The goals of OPNET are to lessen the burden of developing distributed protocol simulations, thus increasing the use of simulation as a supplement or alternative to mathematical analysis of such protocols.

The development of OPNET began in November, 1984, as part of a joint final project by the author and Steven Baraniuk for the MIT Data Communication Networks course (6.263). This initial version allowed the user to set up a packet-switched network with arbitrary point-to-point topology and traffic distributions.

During the summer of 1985, OPNET was extensively rewritten to widen the range of models which the system could support. OPNET is now structured into a three layer modeling system which allows the user to concentrate his efforts on the portion of the models which are relevant to the experiment. The highest layer models issues related to network configuration such as topology, mobility, node and link failure and connectivity. The second layer models issues related to device configuration. Devices are specified to OPNET as networks of standard building blocks called modules. The third and lowest level, allows the insertion of procedures into certain modules.

### 3.2 Overview

Frequently the designer of computer-based systems will find a simulation of either hardware or software helpful, if not necessary, especially as system complexity increases. Some examples of these simulations are instruction-set emulators on development systems and logic-gate simulations on CAE workstations. Such tools allow the designer to test out his code before the hardware is ready, or to test his hardware ideas before actually implementing them, and their use in commercial, research, and educational applications has grown significantly during the past decade.

Increasingly, research and development of distributed systems, such as packet-switched data networks, has relied on simulation. This trend has been driven by:

- 1) Increasing complexity of telecommunications hardware and protocols, bringing with it increased difficulty of mathematical analysis. Specifically, recent trends include getting the most throughput for a given channel bandwidth, raising the number of network customers while keeping delay reasonable, and employing sophisticated data securing and anti-jamming techniques. Frequently, realistic scenarios are beyond the scope of analytic statistical expressions.
- 2) Growing numbers of data network vendors and private data carriers, and the international drive for protocol standardization [3]. Simulation provides a supplement to proof-based verification of designs implementing these standards.
- 3) Success enjoyed by hardware and software emulations in computer development, where almost all sophisticated systems (e.g., devices with greater than 1000 subcomponents) are designed with the aid of CAD tools.

Many simulations of specific protocols have been created, and used to study their behavior under various operating conditions. In addition, large simulations have been created which model the operation of a complete existing or proposed network, on the OSI layers that are of concern to the developers. The amount of parameterization and level-of-detail of each of these

simulations has varied, as have their longevity as useful tools. Needless to say, most of these simulations have a lot of code in common, and a smaller amount of code which is protocol or network-specific. A method of reducing the time and effort required to develop these simulations is to separate the common code into a standard package available to simulation developers. This package would provide a "substrate" upon which the network-specific code could be written. I will refer to such a package as a simulation compiler; its purpose is to accept a condensed specification of the data network structure and operation conditions and generate a ready-to-run simulation.

OPNET is a simulation compiler. Its inputs are abstract specification of protocols and the packet-switched network on which they are executing. Its output is a simulation which can be used to analyze the protocol and network. Several protocol simulation compilers have been proposed or written, but most of these were developed to analyze a certain class of protocols or networks, and thus are much easier to use for that class than for any other. The goal of OPNET is to be easy to use for as wide a range of protocols as can be accommodated by a unified and simple modeling technique.

According to common classifications, the simulations generated by OPNET fall in the following categories:

*Dynamic:* they model a network's state as it evolves over some time interval.

*Stochastic:* they include random variables (e.g. for network load generation) and thus a given run of the simulation represents an estimate of "typical" network performance (specific atypical load conditions can also be applied).

*Discrete:* all finest-grain events in the simulation are associated with a single time interval (the state of the network model changes only at a countable number of points in time).

*Fixed-Increment Time Advance:* time is represented by a variable which

increments by the same amount throughout the simulation. Some simulations use a method called next-event time advance, in which the time variable skips ahead to the time of the next scheduled event, to save procedure calls which would produce no effect. This method was not used because with models as complex as those in OPNET, predicting which event will occur next is as costly (or even more costly) than actually clocking through the unproductive cycles.

*Hierarchical:* OPNET is a hierarchical simulator in the sense that it models a network on several different levels, and gives the user the ability to concentrate on the level most applicable to his needs. For instance, a user interested in network congestion as a function of topology can define a network model using standard, predefined protocols supplied with OPNET. A user interested in testing his own protocols can define them as low-level building-blocks and then create a network model in which to test them.

*Extensible:* After a self-contained part of a network model has been defined, it can be added to the library of "built-in" models, and made available to all users. Thus an installation of OPNET grows more and more customized to the needs of the local users.



### 3.3 Structure

As mentioned previously, an OPNET-generated simulation is quite different from a special-purpose simulation written in a general purpose language such as FORTRAN. In the latter case, the model of the object being simulated is intertwined with routines which support the simulation, such as event sequencing and result analysis code. Like popular simulation languages such as GPSS and Simula, OPNET separates the simulation model from the mechanics of actually running the simulation, determining if the stopping condition has been reached, and collecting and analysing statistics. The low-level simulation management is performed by OPNET system code, and is transparent to the user.

But unlike generic simulation languages, OPNET is designed specifically for simulating data networks. Model primitives familiar to network designers, such as queues and links, are provided so that users can assume them, rather than having to re-implement them. Commonly used operations, such as allocating buffers, searching through queues, etc., are implemented efficiently within the system, and accessible to user models. OPNET will try to provide the efficiency and network-specific features of a special-purpose simulator written in FORTRAN, while retaining the features associated with simulation languages: easy modification, fast development, and the ability to code algorithms clearly and concisely.

Before the beginning of this case study, OPNET was capable of simulating the middle levels of the International Standards Organization (ISO) seven-layer architecture for Open Systems Interconnection. The initial version of OPNET addressed issues of the Network and Data Link layers. Subsequent revisions applied to OPNET during this simulation study (described in section 4.) increased support for simulations of the Physical Layer. Higher layers such as Application, Presentation, and Transport are not simulated by OPNET, due to the fact that they do not have as much to do with network technology as with the application, the computer operating systems involved, etc.

UNIX and C have been chosen as the environment for OPNET, mainly because of their execu-

tion efficiency, good software development tools, and widespread availability at MIT. OPNET simulations are generated by converting the abstract specification of the network into C language programs, which are then compiled into executable object files. Thus OPNET simulations are not much less efficient than similar simulations coded directly in C.

OPNET network models are defined on three separate modeling levels. The top level is called the *network level*. The specifications at this level consist essentially of the interconnection of device models, which represent nodes, via transmission links. The user may create any network topology and use any combination of broadband or baseband links. OPNET can be a powerful tool for network performance evaluation even when used only at this level. In this mode, the user links together predefined device models from the OPNET libraries, and changes various parameters of these models.

The middle level of modeling is called the *device level*. Specifications at this level define node device models, which represent the hardware and software of telecommunications or computing devices (e.g. mainframes with network interfaces, packet switches, multiplexers). Device models are specified in terms of modules, the lowest OPNET building block. Modules of various types can be interconnected using streams to model the functions of the real-world device. A user who models at this level can create devices out of predefined modules, varying the behavior of these modules by changing their parameters.

The lowest level of modeling is called the *module level*. The specifications at this level define the actions of modules, which represent functional abstractions of real-world hardware and software components. Unlike the network and device models, the number of module types is fixed at seven (these types are Processor, Controller, Generator, Queue, Transmitter, Receiver, and Memory). However, the user can fundamentally alter the behavior of the Processor, Controller, Generator, and Queue modules, by writing custom code which conforms to a unified interface. A user who models at this level can access the full flexibility of OPNET, at the expense of learning quite a bit about how modules are designed.

Coding of protocols, custom probability distributions, and custom queue disciplines must now be performed using the C language. The table below summarizes the levels of OPNET modeling, including the compilers used to reduce specification to simulation.

Level	Objects	Interconnects	Language	Compiler
network	devices	links	NSL	netspec
device	modules	streams	DSL	devspec
module	Data Blocks	N/A	C	cc

The basic unit of information flow in OPNET is called the Data Block. Data Blocks can be created with a certain amount of simulated bulk-data, destroyed, enlarged to accommodate control fields, or reduced by discarding control fields. Data Blocks can represent fixed size packets, variable size packets, ascii characters, etc. Any combination of data and control fields is valid. Data Blocks flow from one module to another according to the interconnection set up in DSL, and from one device to another, according to the links set up in NSL.

After each component of an OPNET simulation is written (NSL, DSL, and C files), it must be compiled with the corresponding compiler (as listed in the table above). Devices, modules, and module codes are kept in separate object code files after compilation, so they can be used in multiple simulations. It is the netspec compiler which actually links together all the sub-models and forms the simulation program.

### 3.4 Network Specification

#### 3.4.1 NSL Introduction

Networks are modeled in OPNET by groups of device models (nodes) linked together in an arbitrary-topology graph. Device models are either pulled from a library of predefined, generally useful devices, or created new by the user (see section 3.0 to learn how to create device models). The user obviously requires a way of specifying the configuration of the network model in terms of nodes types and interconnections. The tool provided for this is the *Network Specification Language* (NSL), and its associated compiler, *netspec*. Development of a network model entails:

- 1) The user prepares a file which describes the network model in NSL. This file can be generated using a regular text editor, but its name must end in a ".nf" suffix (nf stands for network file).
- 2) The user must then add a runtime specification to the NSL file. The specification format is detailed in section 5.0 of this manual. For now, assume that it is just more text added to the bottom of the NSL file.
- 3) The user runs the file through *netspec*, the NSL compiler. Assuming no specification errors (NSL syntax or semantic errors) are caught by *netspec*, it will generate the network model as an executable object-code file with the same name as before, except now with a ".sim" suffix. This file is actually a ready-to-run simulation, and can be executed by simply typing its name. The resulting output will be stored in a file with the same name, except with a ".out" suffix (see section 5.0 for explanation of runtime output).

The actual command format for invoking the *netspec* compiler is:

```
netspec file.nf
```

### 3.4.2 NSL Description

A brief description of each NSL construct is given below. All possible options (i.e. parameters, built-in functions) are not detailed, because these frequently change, and are not part of NSL "proper".

#### NSL Programs

An NSL program looks like:

```
ANet =  
  {  
    /* ... (NSL statements) ... */  
  }
```

In this example, ANet is the name given to the NSL-defined network model that is created when netspec is used. Comments can appear anywhere within the braces (comments are denoted by /\* ... \*/, as in C and PL/1). Compiling this program will result in creating the file ANet.sim. Running ANet.sim generates ANet.out, the simulation results file.

#### The Create Statement

The create statement creates instances of a previously defined device, for use within the network model being defined. Create takes a device type argument, as well as a list of names for the created instances (nodes). The device type given must refer to a device that is either part of the system library of devices, or has been previously defined by the user (in DSL) and compiled by devspec. The names supplied in the list must be unique in the whole NSL file. Create will introduce one node of the indicated device type into the network model for each name in the namelist. Subsequent statements can link these nodes together by referring to them by name. An example:

```
create 5ESS {Boston, Chicago, SanFrancisco}
```

#### The Set Statement

The set statement allows the NSL programmer to set parameter values for specific device instances. Set takes a list of device names, as well as list of parameter names and values. The parameters being set are of the device variety, so they tend to deal with node-wide issues such as mobility, etc. Valid device parameters are described in the appendices. An example of the set statement:

```
set {Boston} {fixed}
```

### The Link Statement

The link statement links together nodes that have been previously created using the create statement. Create and link statements can be interleaved in the NSL file, but all nodes referenced in the link statement must have appeared in earlier create statements. The link statement introduces a unidirectional transmission line emanating from a transmitter within the source device and terminating at the receiver of the destination device. Thus nodes to be linked must have transmitter or receiver modules within their device specification. Bidirectional transmissions can be modeled by pairs of reversed link statements. Nodes with multiple transmitters can have multiple outgoing links; likewise, nodes with multiple receivers can have multiple incoming links. Links can also be made between transmitters and receivers in the same node.

A link contains one or more error-free Data Block transmission channels. There are two types of links: baseband and broadband. Baseband links have only one channel, while broadband links may have more than one channel. Both kinds of link have two characteristics: delay and bandwidth. The delay of a link represents the number of iterations it takes for a Data Block to traverse it. The bandwidth of a link is the number of bits per iteration that the link will support. Bandwidth can be separately set for each channel of a broad band link, while delay is the same for each channel.

In order to reference transmitters and receivers within nodes in the link statement, the "transceiver" specification is used. A transceiver is basically just the name of the desired node, followed by the name of the desired transmitter or receiver within that node (this second name

must be in brackets, since it is really a subscript). Thus a transceiver specification for a transmitter (transA) of a node (N1) would look like:

```
N1 [transA]
```

Link takes a pair of transceivers as arguments, separated by an arrow indicating the direction of the link. The next argument is a parameter list, which can set parameters of the link, such as delay, bandwidth, etc. An example of some link statements which create a bidirectional link, using the nodes created before:

```
link Boston [out1] > Chicago [in1]  
  {delay = 20, bandwidth = 500}
```

```
link Boston [in1] < Chicago [out1]  
  {delay = 20, bandwidth = 500}
```



### 3.4.3 NSL Example

Here is an example of a complete NSL program which defines a model of a small network. There are three terminals (device tty) which are multiplexed (mux) and then connected to a mainframe (vax). Note that the computer-to-terminal bandwidth is larger than the terminal-to-computer bandwidth.

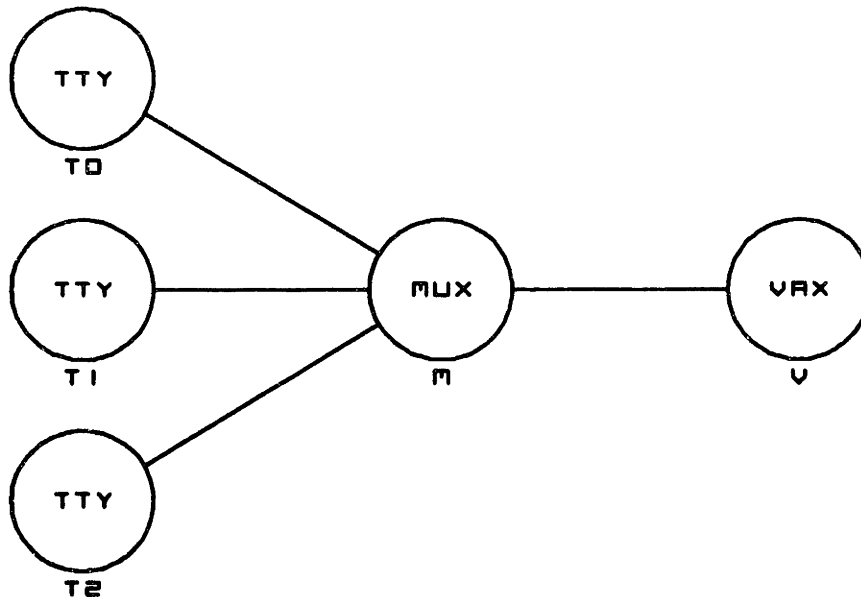
```
net =
{
  create tty {T0, T1, T2}
  create mux {M}
  create vax {V}

  /* terminal to multiplexer links */
  link T0 [out] > M [in0] {bandwidth = 2}
  link T1 [out] > M [in1] {bandwidth = 2}
  link T2 [out] > M [in2] {bandwidth = 2}

  /* multiplexer to terminal links */
  link T0 [in] < M [out0] {bandwidth = 5}
  link T1 [in] < M [out1] {bandwidth = 5}
  link T2 [in] < M [out2] {bandwidth = 5}

  /* multiplexer / computer links */
  link M [muxout] > V [in] {bandwidth = 6}
  link M [muxin] < V [out] {bandwidth = 15}
}
```

A diagram of this NSL-defined network would look like:



### 3.4.4 NSL Syntax

In the following syntactic definitions (loosely based on Backus-Naur Forms), strings within angle-brackets represent instances of syntactic classes, while phrases without these brackets represent literal text. Three periods represent zero or more instances of the last syntactic class. The following syntactic classes are assumed, rather than defined: *<integer>* (a non-negative integer between zero and 32768), *<text>* (an ascii text string), and *<name>* (a name composed of alphanumeric characters (0-9, a-z, A-Z) which must begin with a letter (a-z, A-Z)).

```

<nsl-program>:
  <network-name> =
  {
    <nsl-statement>
    ...
  }

<nsl-statement>:
  <comment>
  <create-statement>
  <link-statement>

<comment>:
  /* <text> */

<create-statement>:
  create <device_name> <name-list>

<link-statement>:
  link <link-pair> <parameter-list>

<link-pair>:
  <transceiver> > <transceiver>
  <transceiver> < <transceiver>

<transceiver>:
  <device-name> [ <module-name> ]

<parameter-list>:
  { <parameter-assign>, ... }

<parameter-assign>:
  <link-parameter> = <parameter-value>

<link-parameter>:
  (see link descriptions for valid parameters)

<parameter-value>:
  (see link descriptions for valid values)
  <integer>

```

```
<name-list>:  
  { <name>, ... }
```

```
<net-name> :  
<device-name> :  
  <name >
```

### 3.5 Device Specification

#### 3.5.1 Introduction

Devices in the context of OPNET are software objects which model the dynamic operation of an actual telecommunications or computing device implemented in hardware or software (or with components of each). Both internal and external characteristics of the device are modeled, but the level-of-detail of the model is determined by the person who designs it, so there are many possible ways to model the same real-world device. In most OPNET simulations, devices are used as nodes in a network, communicating via links. Within such a network, there can be different device types, separate instances of the same device types, or any combination thereof. In some simulations, it is desirable to just look at the internal operations of a single device (e.g., when a device represents a stand-alone parallel computer).

Since devices must be built up from smaller units (modules), the user requires a way of specifying the configuration of his device model in terms of modules and module interconnections. The tool provided for this is the *Device Specification Language (DSL)*, and its associated compiler, *devspec*. Development of a device model entails:

- 1) The user prepares a file which describes the device model in DSL. This file can be generated using a regular text editor, but its name must end in a ".df" suffix (df stands for device file).
- 2) The user runs the file through *devspec*, the DSL compiler. Assuming no specification errors (DSL syntax or semantic errors) are caught by *devspec*, it will generate the model as an object-code file with the same name as before, except now with a ".ob" suffix. This device model can then be used freely in network models.

The actual command format for invoking the *devspec* compiler is:

```
devspec file.df
```

### 3.5.2 DSL Description

The Device Specification Language allows the user to create device models which may then be used in the Network Specification Language to assemble a network model. The devices are constructed using the fundamental building blocks provided by OPNET called modules (see section 4.0 to learn how to specify modules). A program written in DSL must respect the syntax specified in section 3.4.

This section contains descriptions of each DSL construct. All arguments are not discussed here, as they are not part of DSL "proper". DSL syntax has a lot of similarity to NSL, mainly for mnemonic convenience. Statements such as create and set do the same operations in DSL as in NSL, except they create and set the parameters of modules instead of devices. The NSL link statement is not part of DSL, but the equivalent statement is connect (a distinction is made here because the links which join devices together are different in character to the connections which join modules). DSL has other statements, such as domain and external, with no parallels in NSL.

### DSL Programs

A DSL program looks like:

```
ADev =
    {
        /* ... (DSL statements) ... */
    }
```

In this example, ADev is the name given to the DSL-defined device model that is created when devspec is used. Comments appear anywhere within the braces.

### The Create Statement

Create statements allow the DSL programmer to create instances of the modules supported by OPNET and include them in his device model. The module type and the names of the created instances are arguments to each create statement. For example,

```
create queue {q1, q2}
```

will create two queue modules within the encompassing device and assign them the names q1 and q2, respectively. These names may be used in all other references to the created queues, including those in NSL. All the OPNET module types are valid arguments for the create statement.

### The Set Statement

The set statement allows the DSL programmer to set parameter values for specific module instances. The set statement requires module type, module name, parameter names and values as arguments. Valid parameters are described in the manual section on modules. This command is best described by an example:

```
set processor {myprocessor, processor0}
    {incount = 5, outcount = 1}

set receiver {r0, r1}
    {broadband, bandwidth = 10000, channelcount = 4}

set queue {buffer} {accessmethod = fifo}
```

The first statement sets the processors named "myprocessor" and "processor0" to each have five inputs and one output. The second statement sets the receivers named "r1" and "r0" to both be broadband, with total bandwidths of 10000, and four channels each (hence channel bandwidths of 2500). Finally, the third statement sets the queue named "buffer" to be accessed using the FIFO method.

**NOTE:** The DSL compiler will cause an error if an attempt is made to set a parameter of modules which were not created within the current DSL file.

### The Connect Statement

The connect statement is used to setup the data-paths within the device. A single output (or input) from (or to) a module is referred to as a stream. Some modules have only a single input

or a single output stream, while others have many of each. A generator, for instance, has only an output stream, whereas a processor may have an arbitrary number of input and output streams. The connect statement expects pairs of input and output streams as arguments. It has the effect of joining the two streams so that they form a continuous conduit for Data Blocks from one module to another. All data exiting the output stream enters the input stream which is bound to it. For example,

```
connect {queueA > procA, genA > queueA}
```

creates a data path which starts at the output of "genA", flows through "queueA", and ends at the input to "procA."

A module other than a receiver or a generator may not be left unconnected or the device specification compiler (devspec) will generate an error.

### The Domain Statement

The domain statement is used to assign control of a portion of the device to a controller module. A domain is a subset of all the modules within the same device. A controller's domain may not extend beyond the limits of the device defined in the DSL program. OPNET will allow the controller code to manipulate any of the modules within the specified domain. A controller may be set up to change the parameters of any other module, or remove and insert data into the streams of other modules.

The arguments of the domain statement are a controller name list and a module name list. Domain statements have the effect of adding all the modules referenced in the module name list to the domain of each controller referenced in the controller namelist. For example,

```
domain {cont_a, cont_b} {q1, s1, s2}
domain {cont_a} {q2, s3}
```

places modules named "q1", "s1", and "s2" within the domain of the controller named "cont\_b", and the five modules "q1", "q2", "q3", "s2", and "s3" in the domain of controller

"cont\_a".

### The External Statement

The external statement provides a method for controllers within a device to access network wide variables. Its arguments are two name lists: a controller name list and a variable name list. Each controller within the controller name list will be allowed access to variables within the variable name list. Variable names may be arbitrary. However, they must agree with those "bound" in the snapshot statement of the NSL file. See section 5.0 for a description of how these bindings are achieved.

For example,

```
external {cont_a, cont_b} {flow1, flow2}
external {cont_b} {qsize1}
```

allows controller "cont\_b" access to variables "flow1", "flow2", and "qsize1". Controller "cont\_a" has access only to variables "flow1" and "flow2".

#### 3.5.3 DSL Example

An example of a complete DSL program which defines a model of a small device is presented on the next page. The device, called ComplexLoad, includes two generators with radically different load characteristics (bernoulli and poisson distributions) which feed their own queues. The queues are sampled on alternate iterations by the processor, using the QFirst access method. The processor is of type TdmTwo. The output of the processor is fed into a transmitter, resulting in a transmitted output stream which is the sum of the two generated streams.



```

ComplexLoad =
{
/* Generators; note destination will be ignored */
create generator {Gp, Gb}
set generator {Gp} {load = poisson (0.3)}
set generator {Gp} {load = bernoulli (0.05)}

/* Queues */
create queue {Qa, Qb}
set {Qa, Qb} {accessmethod = QFirst}

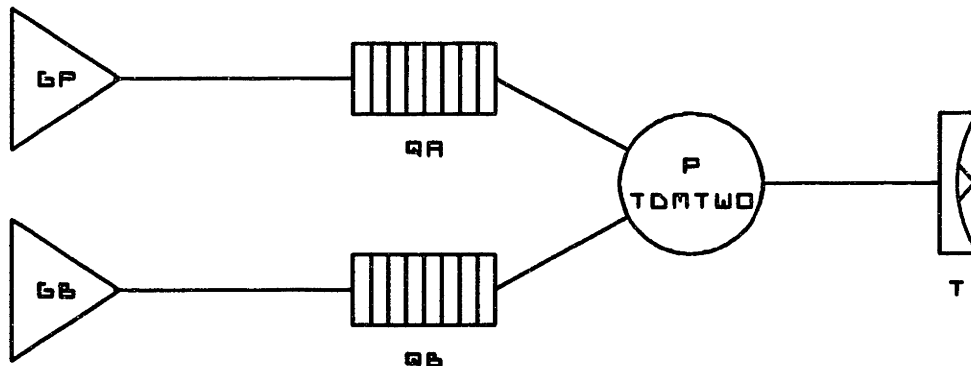
/* Processor */
create processor {P}
set processor {P} {type = TdmTwo}

/* Transmitter */
create transmitter {T}
set transmitter {T} {baseband, bandwidth = 1000}

/* establish connections */
connect
{
Gp > Qa, Qa > P [0], /* poisson queue input */
Gp > Qb, Qb > P [1], /* bernoulli queue input */
P > T
}
}

```

A diagram of this DSL-defined device would look like:



### 3.5.4 DSL Syntax

In the following syntactic definitions (loosely based on Backus-Naur Forms), strings within angle-brackets represent instances of syntactic classes, while phrases without these brackets represent literal text. Three periods represent zero or more instances of the last syntactic class. The following syntactic classes are assumed, rather than defined: *<integer>* (a non-negative integer between zero and 32768), *<text>* (an ascii text string), and *<name>* ( a name composed of alphanumeric characters (0-9, a-z, A-Z) which must begin with a letter (a-z, A-Z), and *<argument>* (devspec parses these as *<text>*; see section 4.0 for valid functions and arguments).

```

<dsl-program>:
    <device-name> =
    {
        <dsl-statement>
        ...
    }

<dsl-statement>:
    <comment>
    <create-statement>
    <set-statement>
    <connect-statement>
    <domain-statement>
    <external-statement>

<comment>:
    /* <text> */

<create-statement>:
    create <module-type> <name-list>

<set-statement>:
    set <module-type> <name-list> <parameter-list>

<connect-statement>:
    connect <connection-list>

<domain-statement>:
    domain <name-list> <name-list>

<external-statement>:
    external <name-list> <name-list>

<parameter-list>:
    { <parameter-assign>, ... }

```

*<parameter-assign>* :  
    *<module-parameter>* = *<parameter-value>*

*<module-parameter>* :  
    (see module descriptions for valid parameters)

*<parameter-value>* :  
    (see module descriptions for valid values)  
    *<integer>*  
    *<name>*  
    *<function>*

*<function>* :  
    (see module descriptions for valid arguments)  
    *<name>* ( *<argument>*, ... )

*<connection-list>* :  
    { *<connection>*, ... }

*<connection>* :  
    *<module-stream>* > *<module-stream>*  
    *<module-stream>* < *<module-stream>*

*<module-stream>* :  
    *<module-name>*  
    *<module-name>* [ *<integer>* ]

*<module-type>* :  
    processor  
    controller  
    generator  
    queue  
    receiver  
    transmitter  
    memory

*<name-list>* :  
    { *<name>*, ... }

*<device-name>* :  
*<module-name>* :  
    *<name>*

#### **4. OPNET Enhancements**

Some of the differences between broadcast and point to point networks were highlighted earlier. The restrictions imposed by OPNET at the network layer previously made it impossible to model broadcast networks. In particular, the network layer expected a rigid topology to be defined in terms node pairs and link parameters. In a radio network or other broadcast network (such as ethernet) the channel is shared among a group of nodes; thus the connectivity criteria between any two nodes has changed. Namely, node A will necessarily transmit to node B if it is in range, and if B is tuned to the same frequency channel as A is transmitting on. Note that is not necessarily a symmetric relationship between the nodes, as one node's transmitter may have more power than the other.

In order to extend OPNET to model broadcast transmission, the transmitter module was given a new parameter indicating the nature of the transmission link (e.g broadcast or point-to-point). If a transmitter is specified to be of broadcast type, upon transmission of a packet, OPNET will compute the connectivity of the transmitter to all receivers in the network and place a replica of the packet at all the receiver sites which are in range. The notion of connectivity assumes that positions and ranges were assigned to the nodes and transmitters. This is done in the network layer model using the *position* and *range* statements.

For full generality, the node positions should be allowed to be time-varying as is often the case for packet radio networks. The mobility of the nodes is currently constrained to be in one of eight directions. (SW, NE, etc..). If a node is specified to be mobile as in the example below, it will randomly reorient itself to travel along one of these eight directions at user-specified time intervals. The nodes are specified to be mobile using the *mobile* parameter, and the network-wide reorientation interval is given by the *mob-interval* parameter.

In addition to correctly placing transmitted packets at all nodes within range, OPNET has been extended to model the packet collision process. At each node, transmissions are assumed to begin at the start of the current iteration. Within that iteration transmitted packets are assigned a starting *bit position* which is the sum of sizes of all the packets already transmitted during the iteration. The

packets carry their bit-positions and packet lengths when they arrive at the destination node. At the destination node packets, the bit-positions of received packets from all nodes are compared to determine which packets collided. The collided packets are destroyed by the receiver. Special care must be taken to also ensure that nodes cannot transmit and receive on the same channel simultaneously. This is done by having transmitters send a special *antenna status* packet to themselves when they transmit. All received packets which intersect intervals during which antenna status packets were transmitted are discarded.

## **5. Case Study**

### **5.1. Background**

In mobile packet radio networks (MPRN), routing may be implemented by organizing the nodes in a tree-like hierarchy [4]. Each node in the hierarchy is aware of the nodes below it, and paths to reach them. The hierarchy must be continuously updated to reflect changes in network connectivity which occur when nodes move.

When node A enters into communication with node B, A's packets proceed up in the hierarchy until they attain a node whose database includes B; they then descend the hierarchy until they reach B itself.

The hierarchy is constructed by flooding special "*probe*" packets from the highest level. When probe packets reach the lowest level, they are returned to the top (through flooding again) recording their path in the process. Each node updates its database from the passing probe packets.

This type of routing strategy was initially proposed for networks of packet radio repeaters whose locations were fixed [5]. In such a case it is clear that the probe mechanism for establishing the hierarchy is not necessary since the topology is available to all nodes. The probes may be performed every so often to reassess the topology if node failures or insertions are anticipated.

However, hierarchical routing methods are applicable to mobile radio networks, so long as the topology assessment probes are conducted at a rate sufficient to keep up with the dynamic network. If the probes are too sparse, packets will be routed incorrectly, in some cases never reaching their destination.

### 5.2. Problem

Hierarchical routing presents a number of difficulties in mobile packet radio networks:

Probing to maintain the network hierarchy consumes valuable bandwidth, and must be performed very frequently in a highly mobile network. Because of its' flooding nature, a probe will induce many packet collisions, therefore taking a considerable amount of time. The collisions will simultaneously increase effective transmission delay of communication already in progress.

Hierarchical routing relies on paths remaining valid between hierarchy updates. Clearly, in a mobile network, this cannot be depended upon, since any particular pair of sequential nodes in a path may move out of each other's range between probes, thereby splitting the path in two.

Many nodes must carry a significant database baggage. The higher level nodes must have a database entry for each node of the network. This is unreasonable for large networks.

It is inherent to hierarchical routing techniques that certain nodes are more crucial than others to network operation. The survivability of the entire network will depend very heavily on the survivability of a handful of nodes at the top levels of the hierarchy. Arrangements must be made for the transfer of database information from a node to its' successor in case of failure. Also, other nodes in the network must be informed that a succession has taken place. The node and its' successor must keep in contact at all times so that a failure can be detected. Thus, hierarchical routing techniques become very complicated when fault tolerance criteria are imposed.

### 5.3. Virtual Circuits in MPRNs

As an alternative to hierarchical routing, a virtual circuit technique adapted to the particular problems of mobile packet radio networks is proposed.

Standard virtual circuits are generally not considered for mobile networks because they depend strongly on topology remaining constant during sessions, since a session uses the same route until it is terminated. However, virtual circuit routing may be employed, if provisions are made to bridge

any gaps in a virtual circuit as soon as they are detected. If a node moves out of range of its' neighbour in a virtual circuit, it may invoke a new *sub-virtual circuit* between itself and its' neighbour, to permit end-to-end communication to continue. This technique is called "*recursive virtual circuit routing*."

This section describes an implementation of virtual circuit routing in detail. The validity of the protocols developed here have been confirmed by simulation.

### 5.3.1. Justification

A number of problems with hierarchical routing in MPRNs have been presented above. Some of these difficulties may be avoided by using recursive virtual circuit routing.

It was pointed out that hierarchical routing necessitated bandwidth-consuming probes of the network topology. Admittedly, virtual circuits also require a flooding process. However, full network floods only occur at session initiation and limited floods are employed to adjust to changes in the virtual circuit connectivity. If the sessions are long and the network is very mobile, then the virtual circuit technique is at an advantage because it requires only one major flood per session, as opposed to the hierarchical routing technique, which requires nearly constant flooding.

We saw that, with hierarchical routing techniques, hierarchy updates must occur at a frequency sufficient to cope with the most dynamic behaviour of the network. If reliable communication is to be ensured at all times, the hierarchy's databases must be constantly valid. In highly mobile, or highly volatile networks (many node outages and link outages possibly due to jamming), reliable communication will require constant network probes; this is unacceptable since probing is a flooding operation. Virtual circuits, on the other hand, adapt to the dynamic topology on an event driven basis. In the steady state (no recent changes in connectivity), the virtual circuits behave as they would in static networks. When a change in connectivity occurs which disrupts a virtual circuit, the network adapts by invoking a sub-virtual circuit which necessitates a new flooding operation. Thus, the routing overhead increases with the topology dynamics rather than being fixed at its' worst case level.



Nodes executing hierarchical routing must carry databases. Virtual circuit routing does not require nodes to keep track of the network topology; each node need only keep track of the virtual circuits which pass through it.

Finally, hierarchical routing strategies create inhomogeneous networks and it follows that network survivability is reduced. When using Virtual circuit (VC) strategies, this problem may be avoided by attributing the same status to all nodes in the network. The failure of any one node will generally not cause major reliability problems in the network.

### 5.3.2 Virtual Circuit Initiation Process

In a mobile network nodes generally have no *a priori* knowledge of the location of other nodes with whom they have been out of communication. One possible solution to this problem is to have a centralized resource maintain network maps, and to have all routing go through this node. This is a special case of hierarchical routing with only two levels in the hierarchy. If it is decided not to use any degree of centralization in the routing process, then the responsibility for locating a target node falls upon the node who wishes to communicate with it.

#### 5.3.2.1 Flooding

Because the position of the target node is arbitrary, only a flooding process will guarantee that it will be reached by a packet requesting communication. Flooding strategies must be slightly modified in packet radio networks because of the broadcast medium. Two problems arise: 1) When a node floods a packet and it is received by one or more neighbours, the packet will be rebroadcast, and the original node will again receive it. 2) There is no way for a node to ascertain how many nodes are within range, and therefore how many should receive the flooded packet, subsequently returning acknowledgements. Thus, the flooding node may not be able to tell whether the flood was successful.

The first problem may lead to exponentially growing floods [4] which will bring the network to a halt if adequate flow-control methods are not used. The difficulty may be resolved in

several ways. One proposed method [4] is to give flooded packets a lifetime measured by the number of node-to-node hops which they have effectuated. This is enough to guarantee that floods will eventually die and that the whole process will affect only the part of the network within a certain radius of the flooding node. Unfortunately, this scheme will still produce wasted retransmissions by the same nodes; each node will retransmit the packet up to  $L/2$  times where  $L$  is the lifetime of the packet when it is first received by the node. In the protocols implemented here, a flooding process which avoided wasted retransmissions was used. This was accomplished by tagging flooded packets with flood-id numbers. When a node receives a packet with the same flood-id as previously, the packet is discarded. Two complications arise when implementing this scheme. At first, only a single flood-id was maintained in the state of the nodes. This allowed two sequential receptions of the same flood to be handled properly. However, when several floods occurred simultaneously in the same region of the network, certain nodes received interleaved flood packets and accepted and retransmitted them, since they appeared to be part of a new flood. This was corrected by maintaining increased state in each node for flood-id's. Since, one node will not, in general, be the source of several simultaneous floods, it is sufficient to maintain one entry for flood-id's per node in the network. It is not sufficient to maintain one entry per neighbouring node, since it is likely that different floods will use the same intermediate node. A second complication arose when a node received two sequential floods from different nodes, but with the same flood-id's. The second flooded packet was rejected. The solution is to associate the flooded packet in the database with the node which spawned the flood. This may be done choosing the flood-id's from a circular space offset by the node identification number times the size of the space, which was the solution adopted for this case study.

The second problem mentioned above has no complete solution. A flooding node will receive acknowledgements for the flooded packets, since the packet-type is transparent to the data-link layer. The data-link layer will assume a successful transmission of the flooded packet if just one ack is received. Subsequent acks will be considered meaningless. The goal of the flood is to reach all neighbouring nodes, which is not guaranteed because of collisions and possible link failure. Thus, some of the neighbouring nodes may acknowledge receipt of the flooded packet while others will not. Although the flood was unsuccessful vis-a-vis certain nodes the data-link layer will not retransmit the

flooded packet. The solution developed here was to force the data-link layer to transmit the flooded packet several times by having the higher layer produce replicas of the packet and sending them down. The number of replicas was referred to in the code as the *flood intensity*. The drawback here is that in some cases the flooding process will use excessive bandwidth and cause a backlog in the node originating the flood. Simulation results seem to indicate however, that all flooded packets did reach their ultimate destination.

### 5.3.2.2 Virtual Circuit Requests

Virtual circuit requests use the flooding mechanism described above to reach the target node. The VC requests initiate at the layers external to the network itself and are first processed by the session layer; at this point they contain only the identification number of the target node.

The goal here is for the target node to receive a VC request packet with a complete description of a valid path linking it to the source node. Thus the VC request packet must accumulate a log of the nodes thru which it passed during its traversal of the network. This log is sufficiently accessed during its return traversal that an appropriate structure for it is a stack, with base and top-of-stack registers. Operations for manipulating the node-id stack are defined in the code for the network and session layer processors (see appendix). They are POP, PUSH, and RESET as used in the traditional meaning. Nodes add their id to the top of the stack with the PUSH(id) operator; they remove the top node-id from the stack with the POP operator; and finally they initialize the stack with the RESET operator. The stack travels with the VC request packet from node to node.

When a node receives a VC request not destined for itself, it PUSHes its own id on to the stack and refloods the packet. When finally a VC request packet arrives at the target node, its stack contains a complete description of the path which it followed. Loops in the path are not possible because the same node would not accept a VC request with the same flood-id twice.

### 5.3.2.3 Virtual Circuit Acknowledgements

When the session layer of a target node receives a VC request packet, it must decide whether or not to accept the VC. In addition, it may receive several VC requests from the same node, only with different routes. This is a result of the flooding process. It must select the best route: in the simulations developed here, the first received packet is assumed to be the best route, as it took the least time to arrive.

If the VC is accepted, a VC acknowledgement packet is generated. The VC request packet is in fact reused since the principle purpose of the stack was to allow the VC ack to return along the selected route while informing all the selected nodes of their participation in the VC. One minor complication arises when considering that, for recursive virtual circuit repair, nodes must know the id of their next-nearest neighbours downstream in the VC. This information, which is available in the stack, is necessary to invoke the sub-virtual circuit as described above. The problem occurs only for the next to last node in the virtual circuit, where there is no next-nearest neighbour; only the target node is left to transmit to in the VC path. The solution adopted here was to view the network and session layers of the target node as two separate hops in the VC path. Thus the target node appears twice at the end of the node-id stack - once for its network layer, and once for its session layer. The next to last node in the VC considers the two nodes downstream to be the target node.

Thus the VC request is converted to a VC ack at the session layer by pushing the target node-id onto the stack (the network layer has already PUSHed it on once), and labeling the packet as a VC ack. The source node's id (which is always affixed to every packet by the network layer) is now the ultimate destination field of the of the VC acknowledgement.

Any node receiving the VC ack can obtain the id's of the two node's downstream by executing the following sequence:

```
second_node_id = POP();
```

```
next_node_id = POP ();
```

```
PUSH (next_node_id);
```

the last PUSH operation is necessary to restore the state of the stack for the next node which will receive the VC ack (the next up-stream node).

In addition to the node\_id's of the next two downstream nodes, each node requires the virtual circuit id of the next node. It is not possible for all nodes to share the same VC id because several VCs may pass thru the same node, and packets arriving during the session may be misrouted if several sessions have common id's [5]. Thus each node selects a unique id for its leg of the virtual circuit and informs the previous node of its choice during the acknowledgement process. The vc-id is simply attached to the VC ack packet before sending it to the next up-stream node.

#### 5.3.2.4. Session Operation

Once the setup process is complete, the virtual circuits function exactly as in point-to-point networks. The only difference is the resolution of the blocking problem. When a node is unable to reach its downstream neighbour (acknowledgements are repeatedly missed), the data-link layer passes a warning to the network layer. The network layer has in its VC tables, the id of the second nearest downstream node. If communication can be established with this node, the one which does not respond may be bypassed. Thus, the network layer invokes a virtual circuit with the next down-stream node by generating a VC request packet with the proper target-id and sending it to the data-link layer. When a VC ack is returned, the nodes in the bypass path are informed of their participation in the new virtual circuit; the hanging branch of the old virtual circuit is eliminated from the routing tables of the last two reachable nodes in the old VC, and replaced with the beginning of the new VC. The last node of the new VC is informed that it is actually not the last node in the VC but rather that the VC is actually continued on with the hanging part of the old VC. Through this process the breach in the old VC is mended. Because the New VC will most likely be very short, the flood of the request packet may be made to be very localized by attributing it a lifetime as described above.

In the steady state operation, packets are labeled with the VC id of the next node in the VC and broadcast. Each node obtains the id and VC id of the next node after itself from its VC tables. The process continues until the packet reaches the target node [5].

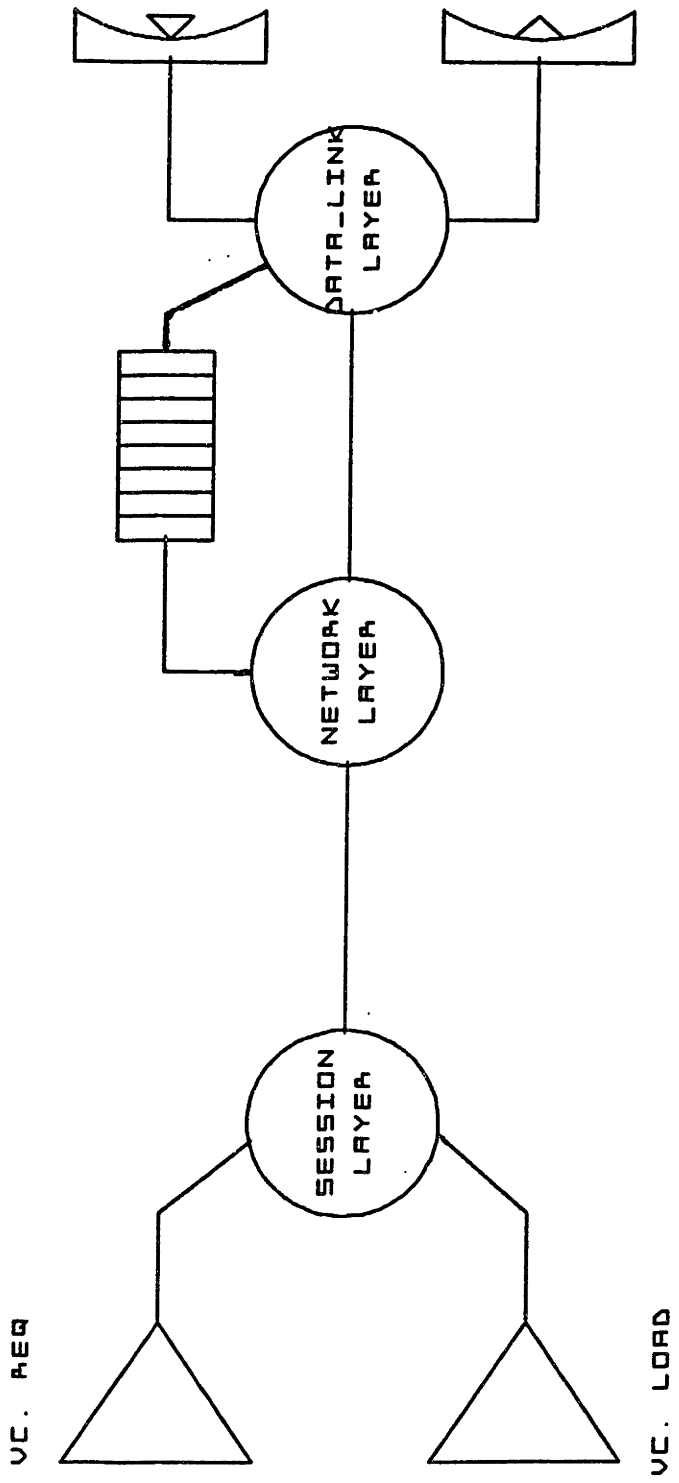


Figure 5.1: Structure of OPNET model for a single node of a Mobile Packet Radio Network.

## 5.4 Models

Figure [5.1] shows the structure of each node in the network in terms of OPNET graphical symbols. Packets transmitted by other nodes in the network within range arrive at the receiver. The lowest layer of the models is the data-link layer, responsible for making the transmission medium appear error-free to the higher layers. The second layer is a network layer protocol unit which keeps track of virtual circuit tables and participates in the VC setup process. New packets are first processed by the session layer processor which maintains statistics and prepares the initial virtual circuit requests and acknowledgements. The session layer processor only processes data which is part of virtual circuits either originating or terminating at the encompassing node. A queue is sandwiched between the data-link layer and the network layer processors, because when retransmissions occur, the data-link layer may not be able to absorb the full contents of the input queue. The generator modules - VC\_REQ and VC\_LOAD- model the application layer which generates virtual circuit session requests and load data for these virtual circuit sessions once they are declared active.

### 5.4.1 Data-Link layer

The process executed by the data-link layer ensures that packets which were lost in transmission are retransmitted. Standard acknowledgement schemes are not quite applicable to packet radio networks because the acks themselves are extremely vulnerable to collisions. Instead, the receiving-end data-link layer processor responds to a received packet with an acknowledgement, but also stores a packet identification number before feeding the packet to the network layer. If the acknowledgement is not received, the data-link layer on the transmitting end will time out and rebroadcast the same exact packet. The receiving end will realize that it has received the same packet twice in a row from the same origin and thus assume that its acknowledgement was lost; thus, it will know to discard the re-broadcast packet to avoid creating duplicates, and also to retransmit the acknowledgement.

Upon time out the data-link layer will not retransmit instantaneously as this would create continuous collisions; all colliding packets would be retransmitted simultaneously, colliding again and again. Rather, a second phase time out is entered whose duration is random. Thus there is a strong likeli-

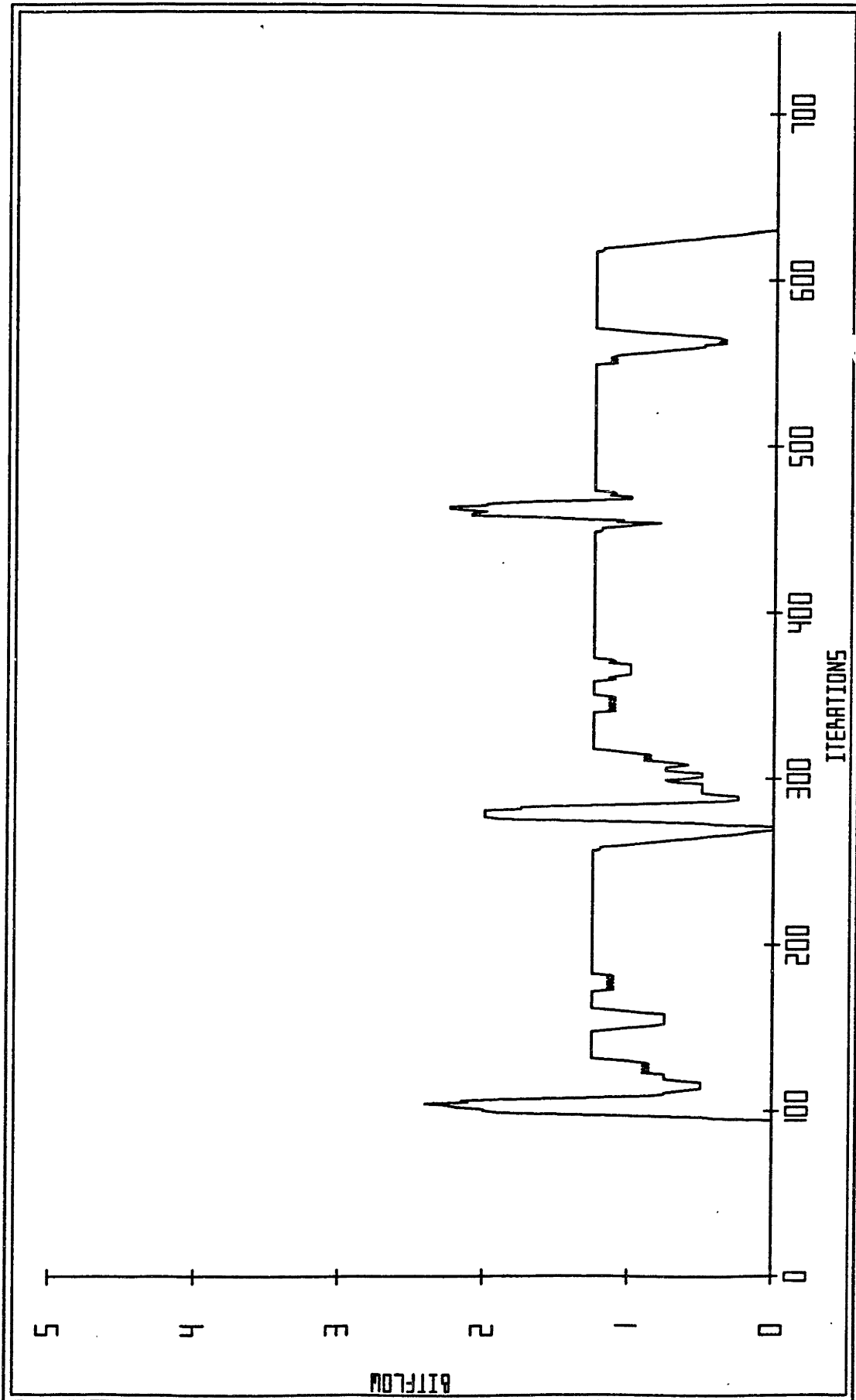


Figure 5.2: Flow out of a node initiating virtual circuit sessions as a function of time.



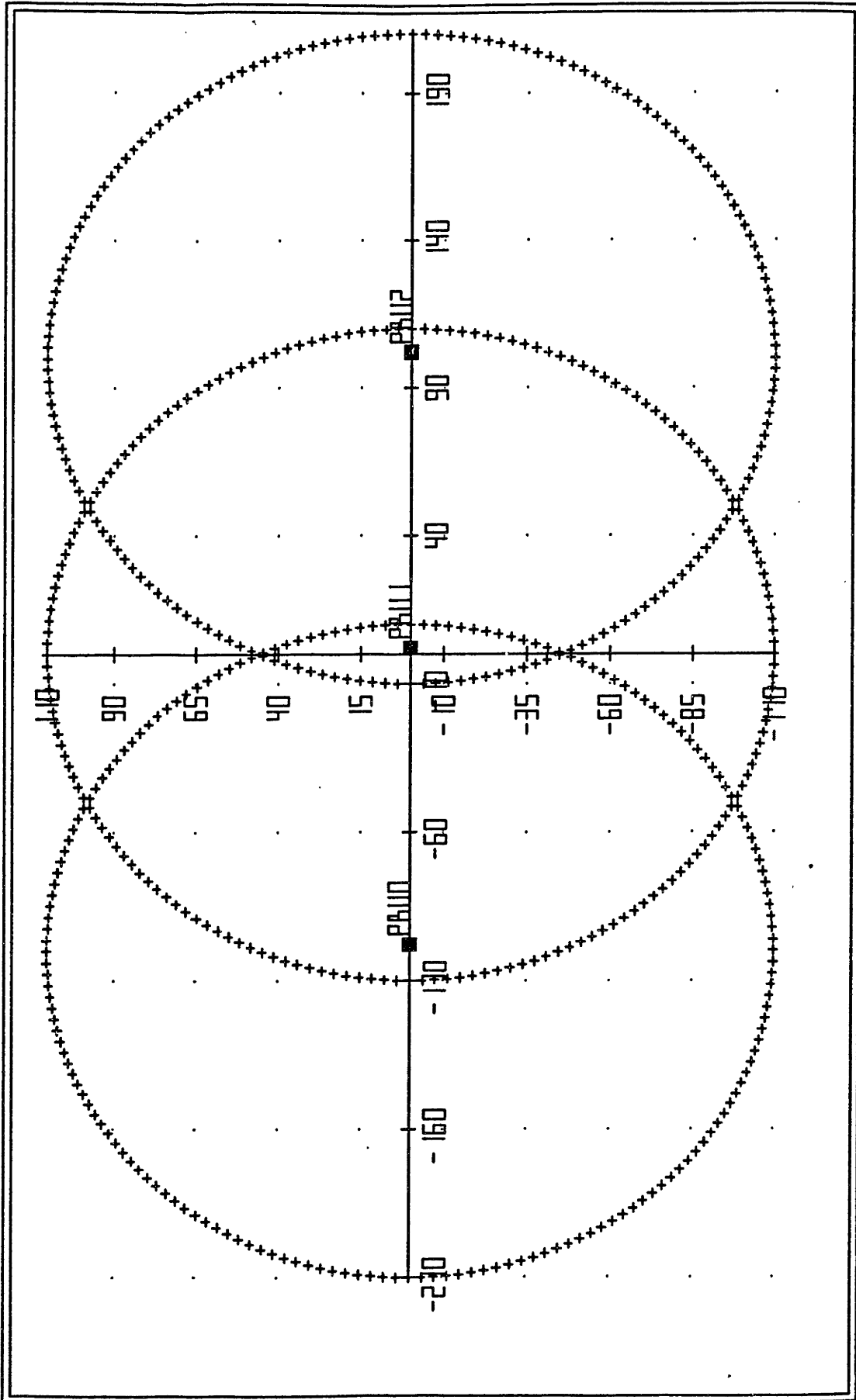


Figure 5.3a: Location of nodes in a three-node network. Only the center node has full connectivity.

node 2	sending a VC request	to	node 0	←
node 0	received a VC request	from	node 2	←
node 2	received a VC acknowledgement	from	node 0	←
node 2	sending a VC termination packet	to	node 0	←
node 2	sending a VC request	to	node 1	←
node 1	sending a VC request	to	node 0	←
node 0	received a VC request	from	node 1	←
node 1	received a VC acknowledgement	from	node 0	←
node 1	received a VC request	from	node 2	←
node 1	sending a VC termination packet	to	node 0	←
node 0	received a vc termination packet	from	node 2	←
node 2	received a VC acknowledgement	from	node 1	←
node 1	sending a VC request	to	node 2	←
node 2	sending a VC termination packet	to	node 1	←
node 0	sending a VC request	to	node 1	←

Figure 5.3b: Listing of VC request, acknowledgement, and termination packets in the three-node network of figure 5.3a.

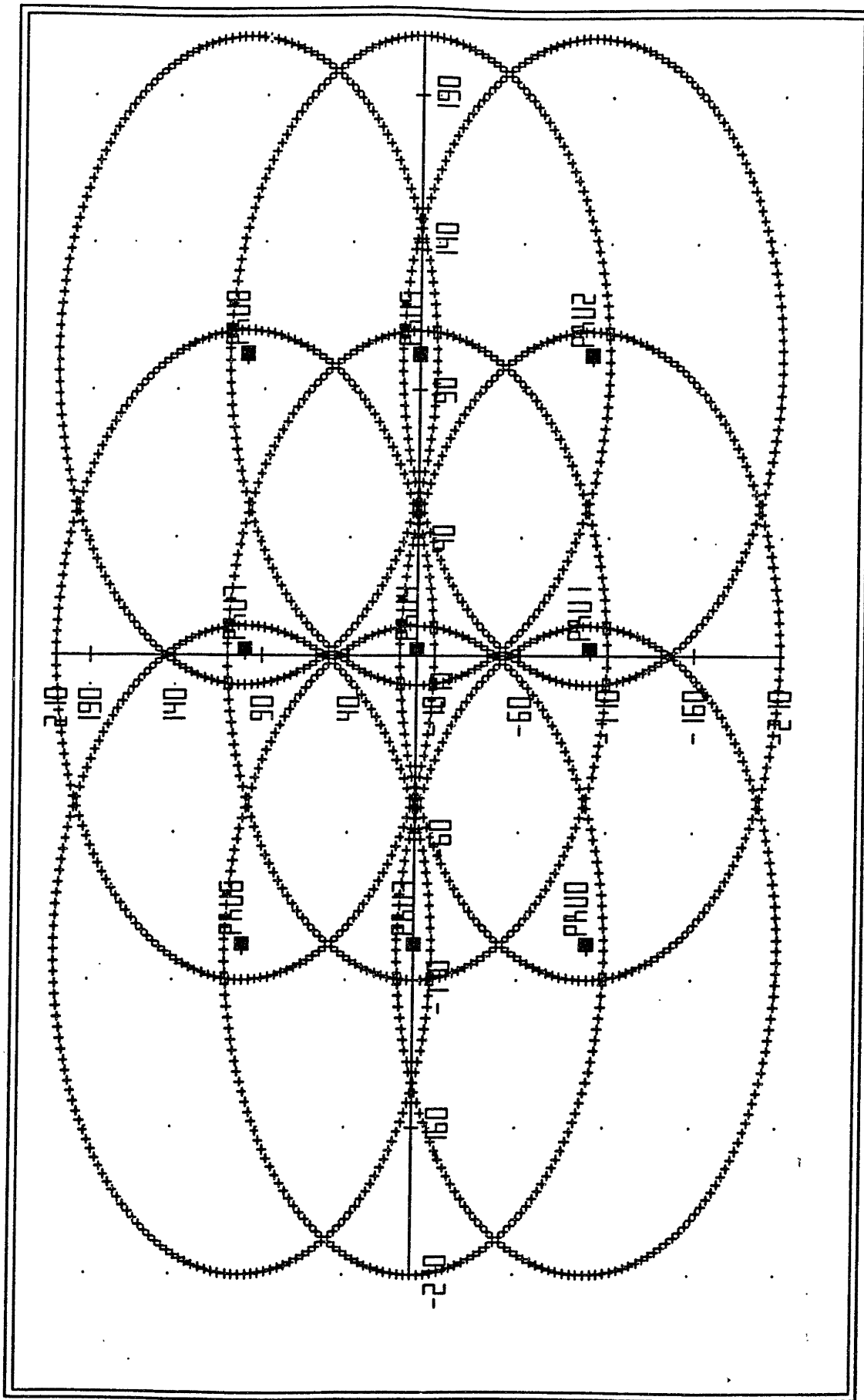


Figure 5.4a: Nine-node network of devices from Figure 5.1.

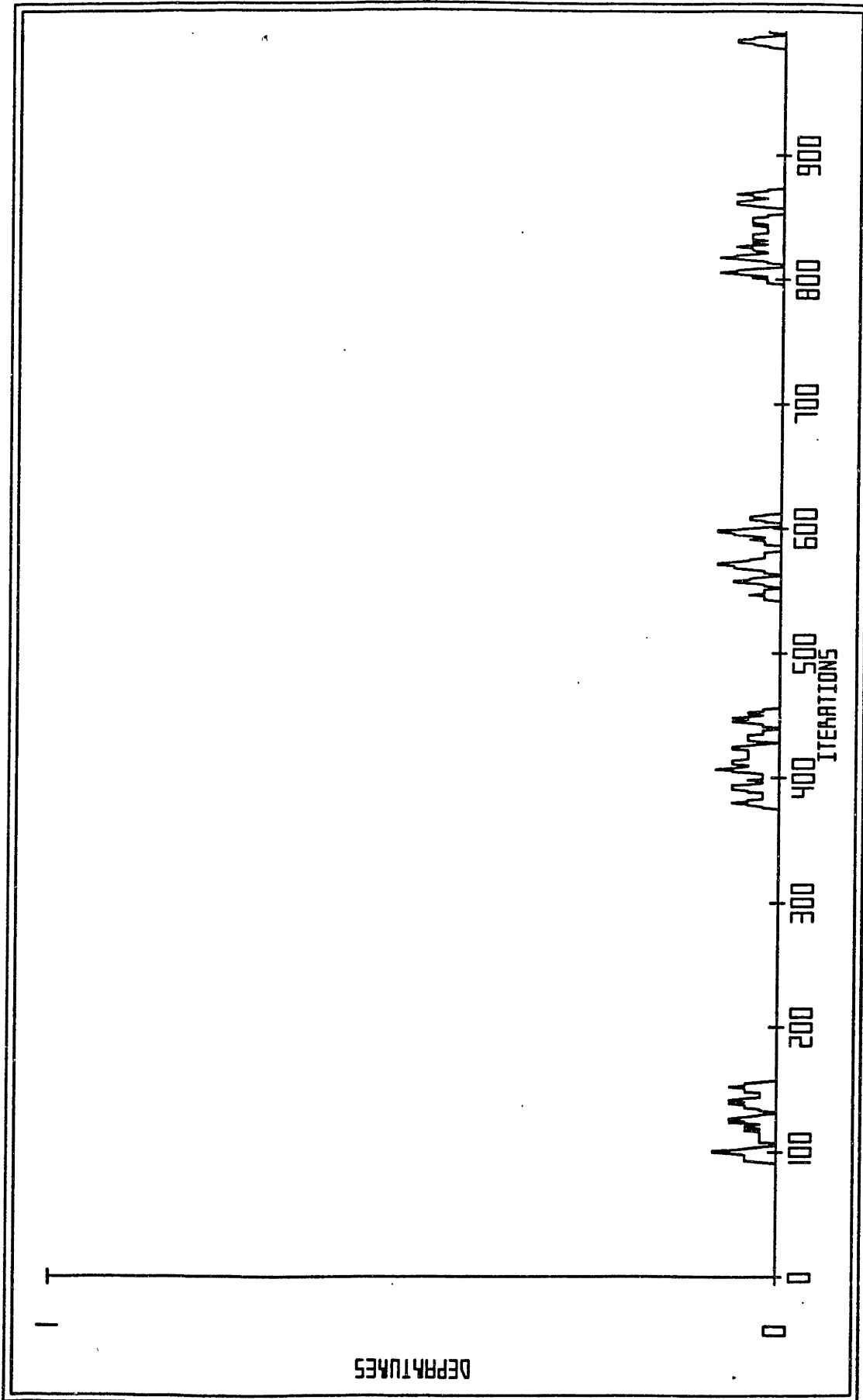


Figure 5.4b: Global average of the number of packets leaving the session layers in the network of Figure 5.4a.

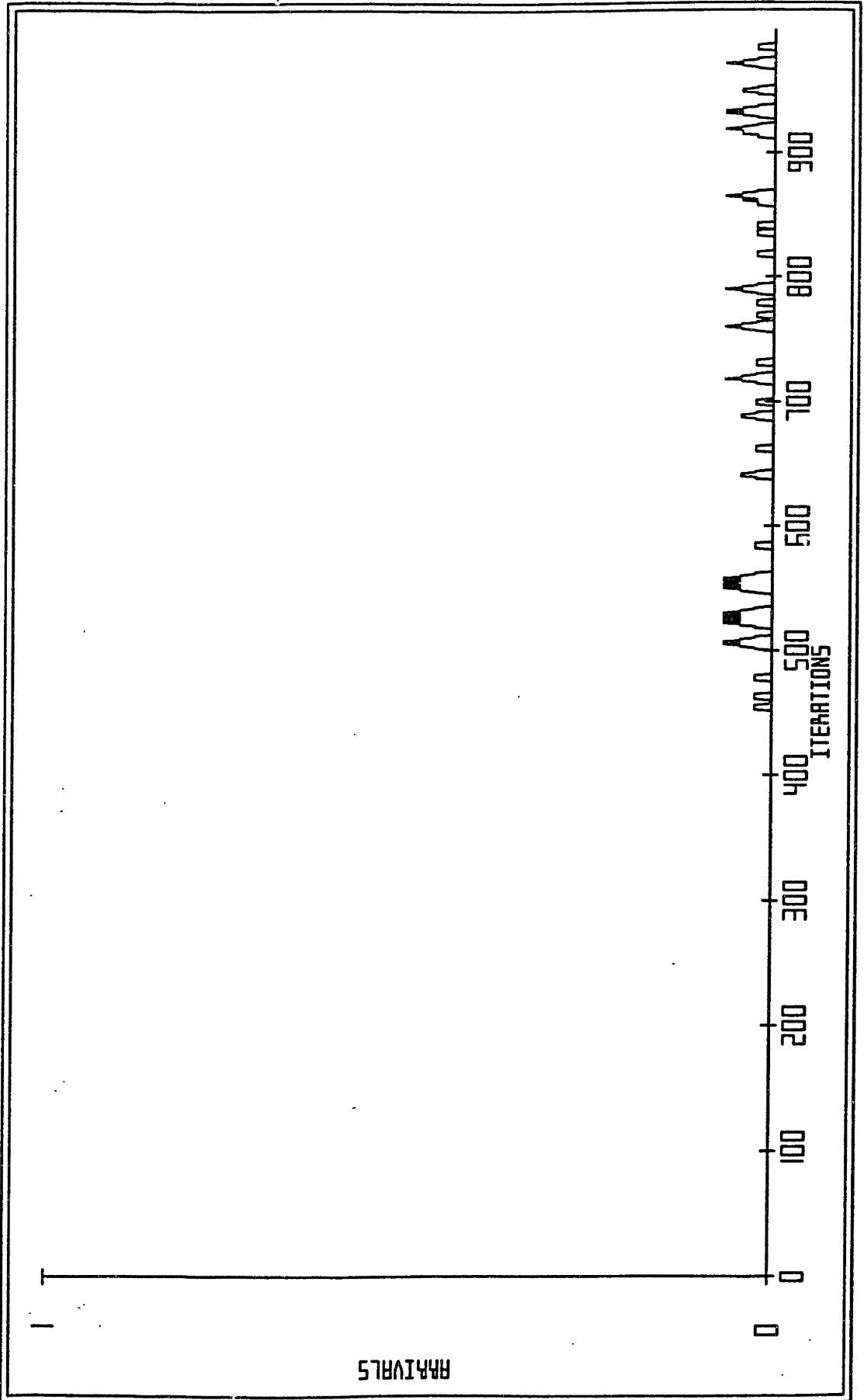


Figure 5.4c: Global average of the number of packets arriving at a session layer after completing a Virtual Circuit traversal.

hood that transmitters will be able to retransmit successfully after a failed transmission. The code executed by the data-link layer processor is given in the appendix.

#### **5.4.2 Network Layer Processor**

The network layer processor implements the recursive virtual circuit protocol. Upon receiving VC requests from the higher layer the net layer processor "pushes" its id onto a stack which will be placed in the VC request packet. It is the first field in the stack. The processor then broadcasts the packet by transmitting it thru the data-link layer.

When receiving a VC request from the data-link layer, the network layer again pushes its id onto the top of the stack. If the VC request is destined for the encompassing node, the network layer feeds it to the session layer processor; otherwise the request is reflooded.

when receiving a VC acknowledgement, the network layer processor pops the top field off the stack and records it as the id of the node which follows it in th VC path. It stores this number in a table along with a unique VC id which it selects. The VC id which it selects is stamped onto the VC acknowledgement and broadcast to the node whose id was just POPped off the stack. The network layer also extracts the unique VC id from the VC ack and stores it in the table: this is the VC id with which to call the next node in the VC path once the session is begun. Network layer code is shown in the appendix.

The session layer in this model is mainly used for formatting initial requests from the load generators and discarding packets which reach their final destination. It also performs some statistic keeping, such as the number of arrivals/departures, and the end-to-end transmission delay of received packets. See appendix for the Session layer processor code.

#### **5.5. Simulation results**

The output of a single node initiating virtual circuit sessions is shown in Figure [5.2] The initial surges in the packet flow are due to the fact that the initiation procedure for a VC session requires

broadcast flooding. Flooding is a difficult process in a PRN because there is no way of knowing how many acknowledgements to expect. Thus, to increase the likelihood that all nodes within range will receive the flooded packet, it is transmitted several times. This is a parameter referred to as `FLOOD_INTENSITY` in the network layer code. This output confirms that the flooding process is being performed, and that broadcasting is functioning since the VCs are acknowledged.

Figure [5.3a] shows a three node network executing the VC protocols. The circles indicate the ranges of the nodes at their respective centers. node PRU0 and PRU2 are not in range of each other. If the VC setup mechanisms work, they should still be able to communicate through node PRU1. This is evidenced by the accompanying listing of Figure [5.3b], in which nodes print out statements of receipt and transmission of VC initiation, acknowledgement, and termination packets. Transmission of the session load takes place between the receipt of the VC ack by the transmitter and the receipt of VC termination packet by the receiver. The VC termination packet is merely routed along the same path as the load data; thus it affects every node in the VC path.

Figure [5.4a] displays a slightly more complicated network executing the same algorithm. The connectivity is more varied, but still all packets departing from the session layer on one end of a VC arrive safely at the session layer on the other as seen figures [5.4a and 5.4b]. This confirms that the underlying mechanisms for packet broadcast, collision detection, and connectivity assessment are functioning properly.

## 5.6 Conclusion

The proper functioning of the low-level mechanisms of OPNET's enhanced capabilities were confirmed by observing that macroscopic characteristics of several models were in accord with the model specifications. Further tests were performed in [2].

Further examination of the virtual circuit algorithms would be necessary to determine their performance under failure modes of the network, and rapidly varying topologies. The overhead associated with the routing should also be examined as a function of the flood-intensity parameter and the maximum number of retransmissions before blockage is declared and a sub virtual circuit is invoked.

**6. References**

- [1] J. Metzner: "On Improving Utilization in ALOHA Networks", in IEEE Trans. Commun., vol. COM-24, pp.447-448, April 1976.
- [2] S. Baraniuk: *MIT Undergraduate Thesis*, MIT Dept. of Electrical Engineering, May, 1986.
- [3] J. D. Day and H. Zimmermann: "The OSI Reference Model," in Proc. of the IEEE, Vol. 71, No. 12, pp. 1334-1340, Dec. 1983.
- [4] I. Gitman, R.M Van Slyke, and H. Frank: "Routing in Packet Switched Broadcast radio Networks," IEEE Trans Commun., vol. COM-24, pp.926-930, Aug 1976.
- [5] A. Tanenbaum: "*Computer Networks*", Prentice-Hall, Inc., New Jersey, 1981.



**7. Appendix**

**7.1 Code Listings** (on following pages)

```

/** layer2.c */
/** code for a data-link processor in packet radio network */

#include <opnet.h>
#include "vc.h"

/** processor input and output channel descriptors */
#define RECEIVE_PHYS 0 /* packets from physical layer arrive here */
#define RECEIVE_NET 1 /* packets from network layer arrive here */
#define SEND_PHYS 0 /* packets to physical layer leave here */
#define SEND_NET 1 /* packets to network layer leave here */

/* data-link layer constants */
#define INACTIVE -1.0 /* database entry is meaningless */
#define TIME_INIT 5 /* timeout interval */
#define ID_WINDOW 100 /* packet id circular space size */
#define NOT_PENDING 0 /* no acks pending */
#define PRE_PENDING 1 /* first time-out not yet elapsed */
#define POST_PENDING 2 /* first timeout elapsed, awaiting second */
#define MAX_POST 10 /* largest second phase of timeout */
#define TO_LIMIT 6 /* max # of timeouts before discarding */

typedef struct
{
    Data_Block held_db;
    int time_held;
    int pending;
    double current_pid;
    double cap_pid [NUM_DEVICES];
    int to_count;
} State;

int extern Time;

data_link ()
{
    State* sptr;
    Data_Block *dbptr0, *dbptr1;
    int i, tpid, node_id;
    double origin, pid, did, in_vc_id;

    /** (0.1) if beginning of operation allocate for and initialize */
    /** state variables of processor */

    if (!Time)
    {
        proc_alloc_state (sizeof (State));
        sptr = (State *) (proc_state_ptr ());
        sptr->time_held = 0;
    }
}

```

```

sptr->pending = NOT_PENDING;
sptr->current_pid = 0.0;
for (i=0; i<NUM_DEVICES; i++) sptr->cap_pid[i] = INACTIVE;
}

```

```

/** (0.2) set pointer to the processor's state, and get the node_id */ 60

```

```

sptr = (State *) proc_state_ptr ();
node_id = proc_dev_id ();

```

```

/** (0) process packets arriving from physical layer */
while (!proc_is_stream_empty (RECEIVE_PHYS))

```

```

{
    dbptr0 = proc_get_first (RECEIVE_PHYS);

```

70

```

    /** if the packet is flooded, disregard the node_id test */
    if (proc_get_field (dbptr0, TYPE_FIELD) == VC_REQUEST)
        goto accept;

```

```

    /** if the packet isnt sent to this node, discard it */
    if (proc_get_field (dbptr0, DEST_FIELD) != (double) node_id)
    {
        proc_destroy_db (dbptr0);
        continue;
    }

```

80

```

accept:
    /** the packet is in the right node: case off its type. */
    switch ( (int) proc_get_field (dbptr0, TYPE_FIELD))
    {

```

```

        case (int) ACK:

```

```

            /** packet is an acknowledgement */
            /** check if it matches held packet */
            if (sptr->pending == NOT_PENDING)
            {
                /* meaningless ack */
                proc_destroy_db (dbptr0);
                continue;
            }

```

90

```

            pid = proc_get_field (&(sptr->held_db),
                                PID_FIELD);
            did = proc_get_field (&(sptr->held_db),
                                DEST_FIELD);
            if ( (proc_get_field(dbptr0, PID_FIELD) == pid)
                && (did==proc_get_field( dbptr0,
                                        ORIGIN_FIELD)) )
                sptr->pending = NOT_PENDING;

```

100

```

            proc_destroy_db (dbptr0);

```

```

            break;

```

```

default: /** packet is not an ACK **/
    origin=proc_get_field (dbptr0,ORIGIN_FIELD);
    pid = proc_get_field (dbptr0,PID_FIELD);
                                                    110

    dbptr1 = proc_create_db (ACK_SIZE);
    proc_set_field (dbptr1, TYPE_FIELD, ACK, 0);
    proc_set_field (dbptr1, ORIGIN_FIELD,
        (double) node_id , 0);
    proc_set_field (dbptr1,DEST_FIELD,origin,0);
    proc_set_field (dbptr1, PID_FIELD, pid, 0);

    proc_output_db (dbptr1, SEND_PHYS);

    /** check if db matches parameters of **/
    /** origin node's entry in data base **/
    /** if entry is inactive, replace it **/
    if (sptr->cap_pid [(int) origin] == INACTIVE)
    {
        /** place in capture database */
        sptr->cap_pid [(int) origin] = pid;
        proc_output_db(dbptr0,SEND_NET);
        continue;
    }
                                                    120

    /** database entry is active: check match */
    /** same as last received */
    if (pid == sptr->cap_pid [(int) origin])
        proc_destroy_db (dbptr0);
    else
    {
        /** place in capture database */
        sptr->cap_pid [(int) origin] = pid;

        proc_output_db(dbptr0,SEND_NET);
    }
                                                    130

    break;
    }

}

/** (1) decrement the timer if pending**/
if (sptr->pending != NOT_PENDING) sptr->time_held-- ;
                                                    150

/** (2) process packets arriving from network layer **/

switch (sptr->pending)
{
    case 0: /** no acknowledgements pending **/
        /** get a packet from the network layer **/

```

```

proc_access_head (RECEIVE_NET);                                160

/** if the network layer gives nothing,leave **/
if (proc_is_stream_empty (RECEIVE_NET))
    {
        break;
    }

/** packet available from the network layer **/
dbptr0 = proc_get_first (RECEIVE_NET);                          170

/** set "pending" flag **/
sptr->pending = PRE_PENDING;
sptr->to_count = 0;

/* transmit the packet to the physical layer */
/* after adding datalink layer info **/

/** test if the local dest_field isnt **/
/** the node itself.. avoiding catastrophes **/
if (proc_get_field (dbptr0, DEST_FIELD) ==
    node_id)
    {
        proc_destroy_db (dbptr0);
        break;
    }

proc_set_field (dbptr0, ORIGIN_FIELD,                          180
               (double) node_id, 0);

/** set the pid field to current value **/
proc_set_field (dbptr0, PID_FIELD,
               sptr->current_pid, 0);

/** increment the pid modulo ID_WINDOW **/
sptr->current_pid += 1.0;
tpid = ((int) sptr->current_pid) % ID_WINDOW;
sptr->current_pid = (double) tpid;                               200

/** install packet as held packet **/
Transfer_Db (dbptr0, &(sptr->held_db));

/* send the packet to physical layer */
proc_output_db (dbptr0, SEND_PHYS);

/** set timer */
sptr->time_held = TIME_INIT;                                    210
break;

```

```

case PRE_PENDING: /* an ack is pending */
if (!sptr->time_held)
{
    /** timed out **/
    sptr->to_count++;
    if (sptr->to_count == TO_LIMIT)                220
        {
            /** exceeded max # of time-outs **/
            /** send a warning to net layer **/
            dbptr0=proc_create_db (TRAN_FAIL_SIZE);
            proc_set_field (dbptr0, TYPE_FIELD,
                TRAN_FAIL, 0);

            /** send with it the number of the vc */
            /** in which the failure occured **/                230

            proc_output_db (dbptr0, SEND_NET);
            sptr->pending = NOT_PENDING;
        }

    else {
        sptr->pending = POST_PENDING;
        sptr->time_held = 1 +
            OS_Random_Limit (MAX_POST);
    }
}                240
    break;

case POST_PENDING:
    /** an ack is pending were in the post-t-out **/
    /** check time_out **/
    if (!sptr->time_held)
        {
            dbptr0=proc_copy_db(&(sptr->held_db));
            proc_output_db (dbptr0, SEND_PHYS);
            sptr->time_held = TIME_INIT;                250
            sptr->pending = PRE_PENDING;
        }
    break;
}
}

```

```

/** layer3.c **/
/** code for a network layer processor in packet radio network **/

#include <opnet.h>
#include "vc.h"

/** processor input and output channel descriptors **/
#define RECEIVE_DL 0 /* packets from data-link layer arrive here */
#define RECEIVE_4 1 /* packets from fourth layer arrive here */
#define SEND_DL 0 /* packets to data-link layer leave here */
#define SEND_4 1 /* packets to fourth layer leave here */

/* network layer constants */
#define FLOOD_INTENSITY 3 /* number of transmissions of a flood packet */
#define MAX_FLOOD_ID 10 /* flood id circular space size */
#define MAX_VC 25 /* max # of vc's */

typedef struct
{
    double next_id; /* next node in the vc path */
    double sec_id; /* node following next one */
    double next_vc; /* vc_id # the next node expects to see */
    double prec_id; /* preceding node id */
    double status; /* VC_SHUT_DOWN or VC_ACTIVE */
} vc;

typedef struct
{
    vc vc_table [MAX_VC]; /* table holding vc information */
    int num_vc; /* number of vc's in action */
    double flood_id [NUM_DEVICES]; /* number of last performed flood */
    int org_flood_id; /* number of last flood originating */
    /* at this processor */
} State;

int extern Time;

#define Push_Vc_List(value) \
    vc_list_ptr = proc_get_field (dbptr0, VC_LIST_PTR); \
    proc_set_field (dbptr0, (int) vc_list_ptr, value, 0); \
    proc_set_field (dbptr0, VC_LIST_PTR, vc_list_ptr + 1.0, 0);

#define Pop_Vc_List(var) \
    vc_list_ptr = proc_get_field (dbptr0, VC_LIST_PTR); \
    var = proc_get_field (dbptr0, (int) (vc_list_ptr - 1.0)); \
    proc_set_field (dbptr0, VC_LIST_PTR, vc_list_ptr - 1.0, 0);

#define Reset_Vc_List_Ptr \
    proc_set_field (dbptr0, VC_LIST_PTR, VC_LIST_BASE, 0);

```

```

network ()
{
    State*      sptr;
    double      node_id;
    Data_Block  *dbptr0, *dbptr1;
    int         i, type;
    double      prec_id, vc_id, dest_id, loc_dest;
    double      o_id, vc_list_ptr, flood_id, temp;
    double      sec_id, next_id, next_vc;
    double      stat_field, ult_dest;
    int         free;

node_id = (double) proc_dev_id ();

/** (0) if beginning of operation allocate and initialize */
if (!Time)
{
    proc_alloc_state (sizeof (State));
    sptr = (State *) (proc_state_ptr ());
    sptr->org_flood_id = 0.0;
    sptr->num_vc = 0;
    for (i=0; i<MAX_VC; i++) sptr->vc_table[i].status=VC_SHUT_DOWN;
    for (i=0; i<NUM_DEVICES; i++) sptr->flood_id[i] = 0.0;
}

    sptr = (State *) proc_state_ptr ();

    /** (1) process packets arriving from data link layer */
    while (!proc_is_stream_empty (RECEIVE_DL))
    {
        dbptr0 = proc_get_first (RECEIVE_DL);
        type = (int) proc_get_field (dbptr0, TYPE_FIELD);

        switch (type)
        {
            case (int) TRAN_FAIL: /* datalink layer suffered failure */
                proc_destroy_db (dbptr0);
                break;

            case (int) DATA: /** packet was ordinary data */
                vc_id = proc_get_field (dbptr0, VC_ID_FIELD);
                o_id = proc_get_field (dbptr0, ORIGIN_FIELD);

                /* check if vc_id-origin pair is active */
                if (sptr->vc_table[(int) vc_id].status == VC_SHUT_DOWN
                    || sptr->vc_table[(int) vc_id].prec_id != o_id)
                {
                    /* not a valid pair */
                    proc_destroy_db (dbptr0);
                    continue;
                }
        }
    }
}

```



```

    }

    /** the vc and origin matched an active vc */
    next_id = sptr->vc_table [(int) vc_id].next_id;
    next_vc = sptr->vc_table [(int) vc_id].next_vc;
    110

    stat_field = proc_get_field (dbptr0, VC_STATUS_FIELD);
    if (stat_field == VC_SHUT_DOWN)
    {
        /* packet is a VC shut down packet */
        sptr->vc_table[(int) vc_id].status=VC_SHUT_DOWN;
        sptr->num_vc --;
    }

    if (next_id == node_id)
    {
        /* packet has arrived at end */
        /* of vc. send higher to layer */
        proc_output_db (dbptr0, SEND_4);
    }

    else
    {
        /* packet is sent to the next */
        /* hop in the virtual circuit */
        130
        proc_set_field (dbptr0, DEST_FIELD, next_id, 0);
        proc_set_field (dbptr0, VC_ID_FIELD,next_vc, 0);
        proc_output_db(dbptr0,SEND_DL);
    }
    break;

    case (int) VC_REQUEST: /** packet is a vc setup request */

        /* if the packet has been flooded and this node has */
        /* already seen it, then throw it away */
        140
        flood_id = proc_get_field (dbptr0, FLOOD_ID_FIELD);
        o_id = proc_get_field (dbptr0, ULT_O_FIELD);
        if (flood_id <= sptr->flood_id [(int) o_id] &&
            flood_id != ((int) o_id)*MAX_FLOOD_ID )
        {
            proc_destroy_db (dbptr0);
            break;
        }

        /* this is the first occ of the flooded packet */
        sptr->flood_id[(int) o_id] = flood_id;

        ult_dest = proc_get_field (dbptr0, ULT_DEST_FIELD);
        /** add the node's id to the node stack */
        Push_Vc_List (node_id);
    150

```

160

```

/* if the vc target is this node, dont reflood */
if (ult_dest==node_id)

```

```

{
    proc_output_db (dbptr0, SEND_4);
}

```

```

else {
    /* flood the packet again */
    for (i = 0; i < FLOOD_INTENSITY; i++)
    {
        dbptr1 = proc_copy_db (dbptr0);
        proc_output_db (dbptr1, SEND_DL);
    }
    proc_destroy_db (dbptr0);
}
break;

```

170

```

case (int) VC_ACK: /* packet is a vc setup ack */
    /* take id off top of vc_list */
    /* store the id along with the next */
    /* node's selected vc_id */

```

180

```

vc_id = proc_get_field (dbptr0, VC_INIT_FIELD);
Pop_Vc_List (sec_id);
Pop_Vc_List (next_id);
Pop_Vc_List (temp); Pop_Vc_List (prec_id);
Push_Vc_List (prec_id);
Push_Vc_List (temp);
Push_Vc_List (next_id);

```

190

```

i=0; while (i< MAX_VC)
{
    if (sptr->vc_table[i].status == VC_SHUT_DOWN)
        {free = i; i= MAX_VC;}
    i++;
}

```

```

sptr->vc_table[free].next_id = next_id;
sptr->vc_table[free].sec_id = sec_id;
sptr->vc_table[free].next_vc = vc_id;
sptr->vc_table[free].prec_id = prec_id;
sptr->vc_table[free].status = VC_ACTIVE;
sptr->num_vc++;

```

200

```

/* assign the vc an id number for this node and let */
/* the previous node know what it is */
proc_set_field (dbptr0, VC_INIT_FIELD,
    (double) free, 0);

```

210

```

proc_set_field (dbptr0, DEST_FIELD, prec_id, 0);

```

```

        if (prec_id == node_id)
            {
                proc_output_db (dbptr0, SEND_4);
            }

        else
            {
                /* send the packet to the previous node */
                proc_output_db (dbptr0, SEND_DL);
            }

        break;
    }

}

/** (2) process packets arriving from layer four */
while (!proc_is_stream_empty (RECEIVE_4))
    {
        dbptr0 = proc_get_first (RECEIVE_4);
        type = (int) proc_get_field (dbptr0, TYPE_FIELD);
        switch (type)
            {
                case (int) DATA: /** layer 4 is sending data */
                    /** as part of a virtual circuit */
                    vc_id = proc_get_field (dbptr0, VC_ID_FIELD);
                    loc_dest = sptr->vc_table[(int) vc_id].next_id;
                    next_vc = sptr->vc_table[(int) vc_id].next_vc;

                    stat_field = proc_get_field (dbptr0, VC_STATUS_FIELD);
                    if (stat_field == VC_SHUT_DOWN)
                        {
                            sptr->vc_table[(int) vc_id].status = VC_SHUT_DOWN;
                            sptr->num_vc --;
                        }

                    /* label the packet so that the next node */
                    /* will recognize the vc id number */
                    proc_set_field (dbptr0, DEST_FIELD, loc_dest, 0);
                    proc_set_field (dbptr0, VC_ID_FIELD, next_vc, 0);

                    /* send the packet to the next node */
                    proc_output_db (dbptr0, SEND_DL);

                    break;

                case (int) VC_REQUEST: /** layer 4 requests a vc */

```

```

sptr->org_flood_id++;
if (!sptr->org_flood_id % MAX_FLOOD_ID) sptr->org_flood_id = 0;
    flood_id = sptr->org_flood_id + node_id * MAX_FLOOD_ID;
    proc_set_field (dbptr0, FLOOD_ID_FIELD, flood_id, 0);
                                                                    270

/* add on to the vc node list */
Push_Vc_List (node_id);

for (i = 0; i < FLOOD_INTENSITY; i++)
    {
        dbptr1 = proc_copy_db (dbptr0);
        proc_output_db (dbptr1, SEND_DL);
    }
proc_destroy_db (dbptr0);
break;
                                                                    280

case (int) VC_ACK: /* layer4 has accepted a vc */
    Pop_Vc_List (prec_id);
    Pop_Vc_List (prec_id);
    Pop_Vc_List (prec_id);
    Push_Vc_List (prec_id);
    Push_Vc_List (node_id);
    Push_Vc_List (node_id);
                                                                    290

i=0; while (i < MAX_VC)
    {
        if (sptr->vc_table[i].status == VC_SHUT_DOWN)
            {free = i; i= MAX_VC;}
        i++;
    }

sptr->vc_table[free].next_id = node_id;
sptr->vc_table[free].sec_id = node_id;
sptr->vc_table[free].next_vc = VC_SINK;
sptr->vc_table[free].prec_id = prec_id;
sptr->vc_table[free].status = VC_ACTIVE;
sptr->num_vc++;
                                                                    300

/* assign the vc an id number for this node and let */
/* the previous node know what it is */
proc_set_field (dbptr0, VC_INIT_FIELD, (double) free, 0);
proc_set_field (dbptr0, DEST_FIELD, prec_id, 0);
proc_output_db (dbptr0, SEND_DL);
break;
                                                                    310
    }
}

```

```

/** layer4.c */
/** code for a session layer processor in packet radio network */

#include <opnet.h>
#include "vc.h"

/** processor input and output channel descriptors */
#define RECEIVE_NET 0 /* packets from network layer arrive here */
#define RECEIVE_REQ 1 /* VC request directives arrive here */
#define RECEIVE_LOAD 2 /* load for vc's arrives here */
#define SEND_NET 0 /* packets to network layer leave here */
#define SEND_SINK 1 /* packets accumulate here */

typedef struct
{
    double life_time;
    double status;
    double next_vc_id;
} vc_cell;

typedef struct
{
    vc_cell vc_set [NUM_DEVICES];
    int num_vc;
} State;

int extern Time;

session ()
{
    State* sptr;
    int i, type;
    Data_Block* dbptr0;
    double node_id;
    double life_time, ult_dest, loc_dest, vc_id;
    double next_vc_id, vc_list_ptr, ult_s;
    double vc_status, o_id;
    int ee_pc, ee_delay_sum;
    double departures, arrivals;

/** macros for manipulating the virtual circuit list */

#define Push_Vc_List(value)
    vc_list_ptr = proc_get_field (dbptr0, VC_LIST_PTR);
    proc_set_field (dbptr0, (int) vc_list_ptr, value, 0);
    proc_set_field (dbptr0, VC_LIST_PTR, vc_list_ptr + 1.0,0);

#define Pop_Vc_List(var)
    vc_list_ptr = proc_get_field (dbptr0, VC_LIST_PTR);
    var = proc_get_field (dbptr0, (int) (vc_list_ptr -1.0));

```

```

proc_set_field (dbptr0, VC_LIST_PTR, vc_list_ptr -1.0, 0);

#define Reset_Vc_List_Ptr
proc_set_field (dbptr0, VC_LIST_PTR, VC_LIST_BASE, 0);

/** (0) if beginning of operation allocate for and initialize **/
/** state variables of processor **/
60

if (!Time)
{
proc_alloc_state (sizeof (State));
sptr = (State *) (proc_state_ptr ());
for (i= 0; i< NUM_DEVICES; i++)
    sptr->vc_set[i].status = VC_SHUT_DOWN;
    sptr->num_vc = 0;
}
70

sptr = (State *) proc_state_ptr ();
node_id = (double) proc_dev_id ();

/** (1) process VC requests arriving from higher layer **/
while (!proc_is_stream_empty (RECEIVE_REQ))
80
{
dbptr0 = proc_get_first (RECEIVE_REQ);
ult_dest = proc_get_field (dbptr0, DEST_FIELD);
life_time = proc_get_bitcount (dbptr0);

/** disallow vc's to the node itself **/
if (ult_dest == node_id) continue;
/** disallow zero-lifetime virtual circuits **/
if (life_time == 0) continue;
90
/** disallow multiple vcs to same node **/
if ( sptr->vc_set[(int) ult_dest].status != VC_SHUT_DOWN)
    continue;

/** the request for a virtual circuit is acceptable **/
/** to the network source **/
/** convert the packet into a VC_REQUEST type packet **/
/** as understood by the network layer **/
100

proc_set_field (dbptr0, TYPE_FIELD, VC_REQUEST, 0);
proc_set_bitcount (dbptr0, VC_REQUEST_SIZE);
proc_set_field (dbptr0, ULT_O_FIELD, node_id, 0);
proc_set_field (dbptr0, ULT_DEST_FIELD, ult_dest, 0);

```

```

    /** initialize the vc list which will build up in the **/
    /** packet as it travels to its ult_dest **/
    Reset_Vc_List_Ptr;
    Push_Vc_List (node_id) ;
    110

    /** send the VC_REQUEST to the network layer **/
    proc_set_field (dbptr0, EE_DELAY, (double) Time, 0);
    proc_output_db (dbptr0, SEND_NET);

    /** initialize the database entry **/
    sptr->vc_set [(int) ult_dest].status = VC_PENDING;
    sptr->vc_set [(int) ult_dest].life_time = life_time;
    printf ("node %d sending a VC request to node %d\n", (int) node_id,
            (int) ult_dest);
    120
}

/** (2) process load data coming in from the application layer **/
/** some of the data isnt apropiate since it is assigned **/
/** to a random destination which may or may not be currently **/
/** linked through a virtual circuit. **/
    130
departures = 0.0;
while (!proc_is_stream_empty (RECEIVE_LOAD))
{
    dbptr0 = proc_get_first (RECEIVE_LOAD);
    ult_dest = proc_get_field (dbptr0, DEST_FIELD);

    /** examine vc database to see if legit dest **/
    if (sptr->vc_set[(int) ult_dest].status == VC_ACTIVE)
    140
    {
        next_vc_id = sptr->vc_set[(int) ult_dest].next_vc_id;
        proc_set_field (dbptr0, TYPE_FIELD, DATA, 0);
        proc_set_field (dbptr0, ULT_O_FIELD, node_id, 0);
        proc_set_field (dbptr0, VC_ID_FIELD, next_vc_id, 0);
        proc_set_field (dbptr0, VC_STATUS_FIELD, (double) VC_ACTIVE, 0);
        proc_set_field (dbptr0, EE_DELAY, (double) Time, 0);
        proc_set_field (dbptr0, ULT_DEST_FIELD, ult_dest, 0);
        proc_output_db (dbptr0, SEND_NET);
        departures++;
    }
    150

    else
    {
        proc_destroy_db (dbptr0);
    }
}
proc_save_variable ("departures", departures);

/** (3) process the packets arriving from the network layer **/

```

```

ee_delay_sum = 0;
ee_pc = 0;
arrivals = 0.0;
while (!proc_is_stream_empty (RECEIVE_NET))
{
    dbptr0 = proc_get_first (RECEIVE_NET);
    type = (int) proc_get_field (dbptr0, TYPE_FIELD);
    ee_delay_sum += (Time - (int) proc_get_field (dbptr0, EE_DELAY) );
    ee_pc++;

    switch (type)
    {
        case (int) DATA: /** data has reached its final **/
            /** destination and may be discarded **/

            /** the packet may be a vc termination packet */
            /** disguised as data.*/
            o_id = proc_get_field (dbptr0, ULT_O_FIELD);
            vc_status = proc_get_field (dbptr0, VC_STATUS_FIELD);
            if (vc_status == VC_SHUT_DOWN)
            {
                sptr->num_vc--;
                printf ("node %d received a vc termination packet from node %d\n",
                    (int) node_id, (int) o_id);
            }

            else
            {
                proc_output_db (dbptr0, SEND_SINK);
                arrivals++;
            }

            break;

        case (int) VC_REQUEST: /** layer 4 must generate a vc_ack */
            Push_Vc_List (node_id);
            proc_set_field (dbptr0, TYPE_FIELD, VC_ACK, 0);
            ult_s = proc_get_field (dbptr0, ULT_O_FIELD);
            proc_set_field (dbptr0, ULT_DEST_FIELD, ult_s, 0);
            proc_set_field (dbptr0, ULT_O_FIELD, node_id, 0);
            proc_set_field (dbptr0, EE_DELAY, (double) Time, 0);
            proc_output_db (dbptr0, SEND_NET);
            sptr->num_vc++;

            printf ("node %d received a VC request from node %d\n",
                (int) node_id, (int) ult_s);

            break;
    }
}

```



```

    case (int) VC_ACK: /* layer4 may install a new vc */
        Pop_Vc_List (next_vc_id);
        ult_dest = proc_get_field (dbptr0, ULT_O_FIELD);
        next_vc_id = proc_get_field (dbptr0, VC_INIT_FIELD);
        sptr->vc_set[(int) ult_dest].next_vc_id = next_vc_id;
        sptr->vc_set[(int) ult_dest].status = VC_ACTIVE;
        proc_destroy_db (dbptr0);
        sptr->num_vc++;
    printf ("node %d received a VC acknowledgement from node %d\n",
            (int) node_id, (int) ult_dest);
        break;
    }
}

if (ee_pc) proc_save_variable ("ee_delay", (double)
                               (ee_delay_sum/ee_pc));
else      proc_save_variable ("ee_delay", 0.0);
proc_save_variable ("arrivals", arrivals);

/** reduce all virtual circuit lifetimes for active vc's */
/** terminate them if their lifetime is zero */

for (i=0; i < NUM_DEVICES; i++)
    if (sptr->vc_set[i].status == VC_ACTIVE)
    {
        if ( ! (--(sptr->vc_set[i].life_time)))
        {
            /** the vc has reached the end of its lifespan */
            /** it will be progressively dismantled from the */
            /** source to the sink */

            /** Note that VC termination packets are disguised */
            /** as ordinary data packets so that they can use */
            /** the same routing logic */

            next_vc_id = sptr->vc_set[i].next_vc_id;
            ult_dest = (double) i;

            dbptr0 = proc_create_db (ACK_SIZE);
            proc_set_field (dbptr0, VC_STATUS_FIELD,
                           (double) VC_SHUT_DOWN,0);
            proc_set_field (dbptr0, TYPE_FIELD, DATA, 0);
            proc_set_field (dbptr0, ULT_DEST_FIELD, ult_dest, 0);
            proc_set_field (dbptr0, VC_ID_FIELD, next_vc_id, 0);
            proc_set_field (dbptr0, ULT_O_FIELD, node_id, 0);

            /** reset the data base entry */
            sptr->vc_set[i].status = VC_SHUT_DOWN;

```

```
    sptr->num_vc--;  
    /** send it to the network layer **/  
    proc_set_field (dbptr0, EE_DELAY, (double) Time, 0);  
    proc_output_db (dbptr0, SEND_NET);  
printf ("node %d sending a VC termination packet to node %d\n",  
        (int) node_id, (int) ult_dest);  
    }  
  
    /** output the number of virtual circuits active **/  
    proc_save_variable ("vc_od", (double) sptr->num_vc);  
  
    }
```

270