INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI®
800-521-0600

An IP/ATM Hybrid Network Simulator – HyNS

Shaozhen Chen

A Major Report
In
The Department
Of
Computer Science

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science

Concordia University

Montreal, Quebec, Canada

September 1998

Abstract

An IP/ATM Hybrid Network Simulator - HyNS

Shaozhen Chen

The IP/ATM Hybrid Network Simulator (HyNS) reported herein is an extension to LBNL's (Lawrence Berkeley National Labs) Network Simulator version 2 -- Ns2 [MCCA97]. Ns2 is an object-oriented simulator, written in C++, with an OTcl interpreter as a front-end. Ns2 has good support for IP, but no support for ATM. Since not many researchers are familiar with Otcl, and it is painful to do OTcl and C++ debugging together, we decided to take out the OTcl part and convert OTcl code into C++ code. In order to support ATM and TCP/IP over ATM, the following components are added to modified and translated Ns2: (1) TCP connection to ATM virtual circuit mapping, (2) second layer routing (ATM switching), (3) ATM switch (Virtual Circuit Classifier), and (4) IP/ATM gateway function (segmentation and reassembly).

HyNS was developed to provide a means for researchers to analyze the behavior of IP networks or IP over ATM networks without the expense of building a real network. It can be used for testing various IP-over-ATM algorithms, investigating the performance of TCP connections over ATM networks without ATM-level congestion control, and comparing it to the performance of TCP over packet-based networks, etc.

Acknowledgments

I would like to thank my supervisor Dr. Manas Saksena for his advice, guidance during the preparation and writing of this report.

I thank my supervisor at work Alf Lund (Software Director) for his support throughout my part time studies, my colleague Alain Beauregard (Senior Software Engineer) for his technical help, Yun Wang (Ph. D. Student) for his help on ATM networking.

Finally, I give my special thanks to my husband for his infinite love and support.

Contents

List of Figures

1. Introduction

1.1 A Brief Introduction of IP Over ATM Networking

1.1.1 Internet Protocol (IP)

Internet is the most popular and the fastest growing network in the world today. The protocols used in the Internet are the Transport Control Protocol (TCP) for end-to-end data transport, and the Internet Protocol (IP) for data transport within the Internet. Most of the Internet today runs on conventional LAN technology like Ethernet and Token Ring for the smaller sub networks and WAN technologies like Frame Relay etc., for long haul links.

1.1.2 Asynchronous Transfer Mode (ATM)

ATM is a connection-oriented network technology that uses small, fixed-sized cells at the lowest layer; ATM was designed to support voice, video, and data with a single, underlying technology. ATM cells of data are transferred through hardware-based switching systems that can work much faster because of the small, fixed-sized cells.

1.1.3 IP over ATM

The deployment of ATM networks is a recent development in the field of computer communication. It seems fairly clear that the telecommunications industry will be deploying ATM infrastructure for the bulk of future high-speed transport, large computer networks will be connection-oriented, with at least the data-link layer connectivity being provided by ATM [KESH95]. These networks will need to communicate with existing networks, the Internet, uses the connectionless IP. In order to take advantage of the high

speed and quality guarantees of ATM with the robust utility of the IP protocol, a number of approaches have been suggested to map Internet Protocol to ATM (IP-over-ATM).

ATM was developed to support many different kinds of digital communications other than simple data transmission. However, data networking is still the most popular use of ATM technology primarily because of vendor focus. ATM's close partnership with synchronous optical networking (SONET) technology has resulted in the development of multi-gigabit delivery systems. Although SONET can work with other networking technologies, ATM/SONET is not only more readily available; it is also a proven environment. Hence, a lot of effort has been put into making data networking protocols such as IP work with ATM. There are a lot of discussions on how to best take advantage of all the features that ATM can provide and incorporate them into IP.

Typical implementation of TCP/IP over ATM uses the following protocol stack [STAL97]:

TCP
IP
AAL5
ATM

ATM Adaptation Layer (AAL) is used to support information transfer protocols not based on ATM, like TCP. AAL Type 5 (AAL5) provides a streamlined transport facility for higher-layer protocols that are connection oriented. In a mixed environment, a convenient way of integrating IP and ATM is to map IP packets onto ATM cells. This will usually mean segmenting one IP packet into a number of cells on transmission and reassembling the packet from cells on reception. The AAL5 layer is responsible for segmentation and reassembling of IP packets that are encapsulated in ATM Protocol Data

Units (PDUs). By allowing the use of IP over ATM, all of the existing IP applications can be employed on an ATM network.

Vendors have decided that IP version 4 can work over ATM by just using that transmission system as a "fat pipe", ignoring the most part of control available with ATM [SHAH97]. The basic connections are established between two end-points with traffic going through large bandwidth pipes of 45 Mbps, 155 Mbps, etc. ATM fat pipes are already taking over WANs for the Internet and corporations.

1.2 The Need for Network Simulator

Network protocols are complex and thus mathematical analysis of their performance is not always feasible. Experimental Studies are not always practical since deployment of new policies is not easy and normally not cost effective. Moreover, not everyone has access to a network that can be modified. Network simulators provide a flexible way of studying network behaviors and performance. Simulation is often the method of choice for evaluating the performance implications of policies and mechanisms on hosts and routers.

1.3 IP/ATM Hybrid Network Simulator (HyNS)

There are many simulators designed for either IP or ATM networks, but few simulators designed for both (one exception is [MAH96]). There is evidence that researchers have to use two different simulators to compare the performance of IP network and IP-over-ATM network [ROMA94]. There are two disadvantages of using two simulators for comparison. First, researchers need to learn two different simulators; second, different simulators will have different implementation for the basic network mechanisms, thus

will have different overhead. Less common background will be there for performance comparison. A dual or hybrid network simulator will be very useful for testing various IP-over-ATM algorithms, investigating the performance of TCP connections over ATM networks without ATM-level congestion control, and comparing it to the performance of TCP over packet-based networks, etc.

Because of the lack of combined IP and ATM simulators, my goal was to develop a dual or hybrid network simulator. In order not to reinvent the wheels, the best way is to combine existing IP and ATM network simulators into a single hybrid simulator. But normally different simulator will have different data and class structure, different communication method between command and configuration interface and simulation engine, and implemented in different languages. It is almost impossible to merge two simulators into one without fully understanding both and modifying at least one of the two extensively. Thus, there is overhead of learning two existing simulators. The more practical way is to choose a best available simulator as the base simulator to modify and add missing functionality to it.

In order to find a good base for developing a new IP/ATM Hybrid Network Simulator, several existing simulators are studied. Lawrence Berkeley National Laboratory (LBNL)'s Network Simulator version 2 (Ns2) [MCCA97] turned out to be the winner. Following are the reasons why Ns2 was chosen as the base simulator:

- Ns2 has good support for IP, it provides substantial support for simulation of TCP, routing, and multicast protocols.
- Ns2 source code is available on the Internet and daily snapshot is provided.

- Ns2 development effort is an ongoing collaboration with the Virtual InterNetwork Testbed Project (VINT)[KUMA97], Ns2 is the simulator code basis for VINT project.
- Ns2 has fine-grain object decomposition; objects can be composed and re-used.
- A lot of notes and documentation is available.
- There are mailing lists for discussions among Ns2-users, for announcements about Ns2.
- Many researchers use Ns2.

Ns2 has good support for IP, but no support for ATM. Since Ns2 is written in C++ and Object Tool Command Language (OTcl), any one who wishes to fully understand or modify it must be comfortable with both Otcl and C++ tools. It is hard to debug both C++ and OTcl code. OTcl also brings overhead in simulation. Since not many researchers are familiar with OTcl, we decided to take out the OTcl part and convert OTcl code into C++ code, this way, user only needs to know C++ to understand the simulator and can step into each line of code by using only C debugger. Thus, there were two main parts of the project:

1. Converting OTcl code to C++ (for a large subset of Ns2).
2. Extending this modified version to support ATM virtual circuits (VCs).

HyNS was originally developed under Unix Solaris 2.5, and ported to Windows NT 4.0 and Windows 95. Nowadays, personal computers can be found every where, in the office or at home. So it is very convenient for the researchers to have a version of HyNS running under Windows environment like Windows NT and Windows 95. See appendix

A for the details of creating HyNS console program using Microsoft Visual C++ 5.0 under Windows NT or Windows 95 environment.

Since not everything was feasible to do in the scope of the project, following features are considered as future work:

- Configuration files

  HyNS uses program *main()* function to setup topology and run simulation. Any configuration changes will mean recompiling the simulator. This is not convenient when the simulator is used for running more than one set of simulations. The better way is to include some initialization code that reads the configuration files.

- ATM Quality of Service (QOS) Control

  HyNS only uses ATM transmission system as a "fat pipe", ignoring the QOS control available with ATM.

- Multiplexing more than one connection into one virtual circuit

  Only one TCP connection per virtual circuit mapping is implemented in HyNS, multiplexing more than one connection into one virtual circuit can be added when needed.

1.4 Outline of this Report

This section contains a roadmap of the remainder of this report.

Chapter 2 IP Network Simulation

This chapter describes the basic IP network simulation concepts mostly inherited from Ns2. A lot of classes and their member functions mentioned in this chapter are the modified and translated version of Ns2.

Chapter 3 OTcl/C++ Conversion

This chapter describes what I did to take out OTcl part and to translate OTcl code to C++ code.

Chapter 4 IP over ATM Network Simulation

This chapter describes what was added to Ns2 to support ATM and IP over ATM network simulation.

Chapter 5 A Simulation Example

This chapter explains the usage of the hybrid network simulator by using a simple example.

Chapter 6 Summary

This chapter summarizes the report.

2. IP Network Simulation

Ns2 is an event-driven network simulator. The simulation engine is written in C++ that uses MIT's OTcl as the command and configuration interface. This chapter describes the basic concepts in the simulation of IP networks, and used in Ns2. A lot of classes and their member functions mentioned in this chapter are modified and translated version of Ns2. The translated class definitions is listed in detail in section 3.3.3.

2.1 Simulator Initialization

When a new simulation object is created, the following operations are performed:

- initialize the packet format
- create a simple linked-list scheduler
- create a "null agent" (a discard sink used in various places)

The packet format initialization sets up field offsets within packets used by the entire simulation. The scheduler runs the simulation in an event-driven manner and may be replaced by alternative schedulers that provide somewhat different semantics, like *heap scheduler* and *calendar scheduler*. The null agent is generally useful as a sink for dropped packets or as a destination for packets that are not counted or recorded.

2.2 Network Topology Generation

A network topology is realized by using three primitive building blocks: nodes, links and agents. The *Simulator* class has functions to create or configure each of these building blocks.

2.2.1 Nodes

Three types of nodes are defined in HyNS, namely IP nodes (IP hosts and routers), ATM switches and gateways. IP node is inherited from Ns2; ATM switch and gateway are new extensions to HyNS. Each node has a unique node ID to uniquely identify each node in the topology; the IDs of IP nodes also represent their addresses. Passing an integer node type parameter to the Simulator function *node (int node_type)* creates IP nodes, ATM switches or gateways, and a unique ID is automatically assigned to each node.

2.2.2 Links

Links are created between nodes to form a network topology with the *simplex_link()* and *duplex_link()* functions that set up unidirectional and bi-directional links respectively. The following describes the syntax of the *simplex_link*() which creates a link from node1 to node2 with specified bandwidth and delay characteristics. The link uses a queue of type queue_type:

> *void simplex_link(Node* node1, Node* node2, double bandwidth, double delay, int queue_type)*

The function *duplex_link*() constructs a bi-directional link from two simplex links.

2.2.3 Agents

Agents are the objects that actively drive the simulation. Agents can be thought of as the processes and/or transport entities that run on nodes that may be end hosts or routers. Traffic sources and sinks are all examples of agents. Once the agents are created, they are attached to nodes with the *attach_agent() Simulator* function. Each agent is automatically assigned a unique port number across all agents on a given node (analogous to a TCP or UDP port). Some types of agents may have sources attached to them while others may generate their own data. For example, you can attach "ftp" and "telnet" sources to "tcp"

agents but "constant bit-rate" agents generate their own data. Sources are attached to agents using the *attach_source() Agent* function.

2.3 Nodes and Packet Forwarding

The function *Simulator::node()* constructs a node out of simpler classifier objects. The typical structure of an IP node is as shown in Figure 1. ATM switches and gateways will be explained in chapter 4.

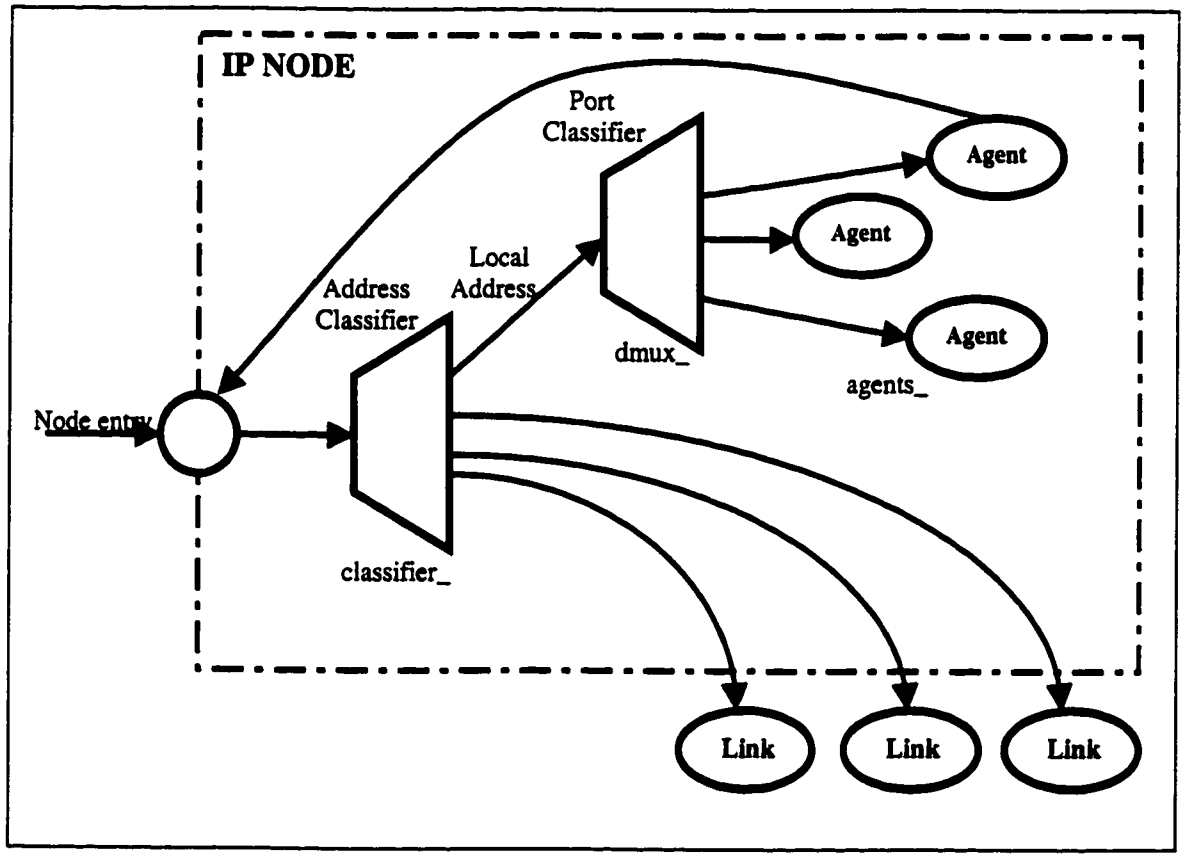

Figure 1 Typical Structure of an IP Node

2.3.1 Node Member Functions

The definition of *Node* class can be found in section 3.3.3.3. Its member functions can be classified into:

- **Control Functions**

entry() returns the entry point for a node. This is the first element that will handle packets arriving at that node. For unicast nodes, this is the address classifier that looks at the higher bits of the destination address. The member variable, *classifier_* contains the reference to this classifier.

reset() resets all the agents at the node.

- **Address and Port Number Management**

id() returns the node ID of the node. This ID is automatically incremented and assigned to each node at creation by the function *Simulator::node()*. The class *Simulator* also stores an array member variable *Node_*, indexed by the node ID, and contains a reference to the node with that ID.

agent(int port) returns the handle of the agent at the specified port. If no agent at the specified port number is available, the function returns the null string.

alloc_port() returns the next available port number. It uses an instance variable, np_, to track the next unallocated port number.

add_route(int dst, NsObject* target) is used by unicast routing to populate the *classifier_*.

add_switch(int vci, NsObject* target) is used by unicast switching to populate the *classifier_*.

- **Agent management**

attach(Agent* agent) will attach the agent to the node, assign a port number to the agent and set its source address, set the target of the agent to be its *entry()*.

Detach() will remove the agent from the node, and point the agent's target, and the entry in the node dmux_ to nullagent.

- **Tracking Neighbours**

Each node keeps a list of its adjacent neighbors in its member variable, *neighbor_*.

add_neighbor() adds a neighbor to the list, *neighbor_*.

neighbors() returns the list, *neighbor_*.

2.3.2 Classifiers

The function of a node when it receives a packet is to examine the packet's fields, usually its destination address. It should then map the values to the next downstream recipient of this packet. A simple classifier object performs this task. A node in HyNS uses many different types of classifiers for different purposes. A classifier provides a way to match a packet against some logical criteria and retrieves a reference to another simulation object based on the match results. Each classifier contains a table of simulation objects indexed by slot numbers. The job of a classifier is to determine the slot number associated with a received packet and forward that packet to the object referenced by that particular slot. The class *Classifier* provides a base class from which other classifiers are derived.

```
class Classifier : public NsObject {
public:
        Classifier();
        ~Classifier();
        void recv(Packet*, Handler* h = 0);
        int maxslot() const { return maxslot_; }
        NsObject* slot(int slot);
        NsObject* find(Packet*);
        virtual int classify(Packet *const);
        void install(int index, NsObject*);
        void print_mapping_table();

protected:
        void clear(int slot);
        void alloc(int);
```

```
        NsObject** slot_;     // table that maps slot number to a NsObject
        int nslot_;           // allocated max slot
        int maxslot_;         // actual max slot used
        int offset_;          // offset for Packet::access()
        int shift_;
        int mask_;
        int off_ip_;
};
```

The function *classify()* is pure virtual, indicating the class *Classifier* is to be used only as a base class. The *alloc()* function dynamically allocates enough space in the table to hold the specified number of slots. The *install()* and *clear()* functions add or remove objects from the table. The *recv()* function and *find()* function are implemented as follows:

```
// objects only ever see "packet" events, which come either
// from an incoming link or a local agent (i.e., packet source)
void Classifier::recv(Packet* p, Handler*)
{
        NsObject* node = find(p);
        if (node == NULL) {
                Packet::free(p);
                return;
        }
        node->recv(p);
}

// perform the mapping from packet to object, return NULL if no mapping
NsObject* Classifier::find(Packet* p)
{
  NsObject* node = NULL;
  int cl = classify(p);
  if (cl < 0 || cl >= nslot_ || (node = slot_[cl]) == 0)
        return (NULL);
       else
        return (node);
}
```

When a classifier receives a packet with function *recv()*, it hands it to the *find()* function to perform the mapping from packet to object, which passes the packet to the *classify()*

function. *classify()* function is defined differently in each type of classifier derived from the base class. The usual format is for the function *classify()* to determine and return a slot index into the table of slots. If the index is valid, and points to a valid object, the classifier will hand the packet to that object using that object's *recv()* method.

The *clear(int slot)* function clears the entry in a particular slot. The *slot(int slot)* function returns the object stored in the specified slot. The *install(int index, NsObject* object)* function installs the specified object at the slot index.

2.3.3 Address Classifiers

An address classifier is used in supporting unicast packet forwarding. It applies a bit-wise shift and mask operation to a packet's destination address to produce a slot number. The slot number is returned from the classify() method. The class *AddressClassifier* is defined as follows:

```
class AddressClassifier : public Classifier {
public:
        AddressClassifier() : mask_(~0), shift_(0) {
          // set up classifier as a router (i.e., 24 bits for addr and 8 bits for port)
          mask_ = AddressClassifier_mask_;
          shift_ = AddressClassifier_shift_;
        }

        nsaddr_t mask_;
        int shift_;

protected:
        int classify(Packet *const p) {
                hdr_ip* h = (hdr_ip*)p->access(off_ip_);
                return ((h->dst() >> shift_) & mask_);
        }
};
```

The class imposes no direct semantic meaning on a packet's destination address field. Rather, it returns some number of bits from the packet's *dst_* field as the slot number used in the *Classifier::recv()* method.

2.4 Links and Simple Links

The class *Link* provides a few simple primitives. The class *SimpleLink* is the subclass of class *Link* that provides the ability to connect two nodes with a point to point link. As with the node being composed of classifiers, a simple link is built up from a sequence of connectors. Figure 2 shows the composite construction of a unidirectional link.

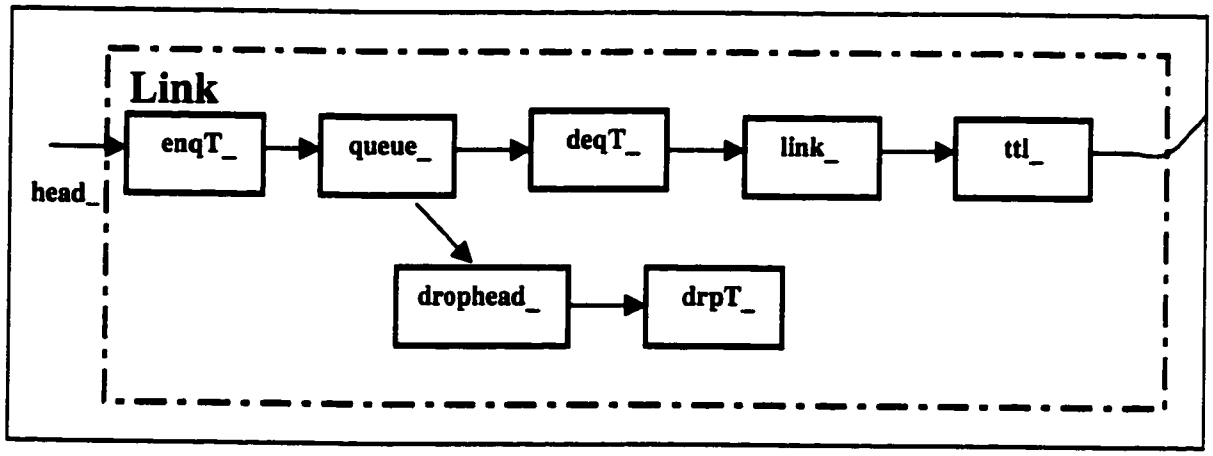

Figure 2 Composite Construction of a Unidirectional Link

Five instance variables define the link:

head_ Entry point to the link, it points to the first object in the link.

queue_ Reference to the main queue element of the link. Simple links usually have one queue per link.

link_ A reference to the element that actually models the link, in terms of the delay and bandwidth characteristics of the link.

ttl_ Reference to the element that manipulates the ttl in every packet.

drophead_ Reference to an object that is the head of a queue of elements that process link drops.

The following member variables track the trace elements:

enqT_ Reference to the element that traces packets entering queue_.

deqT_ Reference to the element that traces packets leaving queue_.

drpT_ Reference to the element that traces packets dropped from queue_.

2.4.1 Link Member Functions

The following are the member functions of class *Link:*

head() returns the handle for head_.

queue() returns the handle for queue_.

link() returns the handle for the delay element, link_.

cost() set link cost to value specified.

get_cost() returns the cost of the link. Default cost of link is 1, if no cost has been specified earlier.

2.4.2 Class Connector

Connectors only generate data for one recipient; either the packet is delivered to the *target_* neighbor, or it is sent to the *drop_target_*.

A connector will receive a packet, perform some function, and deliver the packet to its neighbor, or drop the packet. There are a number of different types of connectors, each performs a different function.

LinkDelay Object that models the link's delay and bandwidth characteristics. This object schedules receive events for the downstream object for each packet it receives at the appropriate time for that packet.

Queue models the output buffer attached to a link in a "real" router in a network. In HyNS, It is attached to, and is considered as part of the link.

TTLChecker will decrement the ttl in each packet that it receives. If that ttl has a positive value, the packet is forwarded to the next element on the link. In the simple links, *TTLCheckers* are automatically added, and are placed as the last element on the link, between the delay element and the entry for the next node.

2.5 Agents

Agents represent endpoints where network-layer packets are constructed or consumed, and provide some functions helpful in developing transport-layer and other protocols. Generally, a user wishing to create a new source or sink for network-layer packets will create a class derived from Agent.

2.5.1 Agent State

The class *Agent* includes enough internal state to assign various fields to a simulated packet before it is sent. This state includes the following:

*addr_* source node address

*dst_* destination node address

*size_* packet size in bytes (placed into the common packet header)

*type_* type of packet (in the common packet header)

*fid_* the IP flow identifier

*prio_* the IP priority field

*flags_* packet flags

*deftl_* default IP ttl value

Any class derived from Agent may modify these variables, although not all of the agents need all of the variables.

2.5.2 Agent Member Functions

The class *Agent* supports packet generation and reception. The following member functions are generally not over-ridden by derived classes:

Packet allocpkt()* allocates new packet and assigns its fields

Packet allocpkt(int)* allocates new packet with a data payload of n bytes and assigns its fields

Classes derived from *Agent* should normally overrride following member functions:

void timeout(int timeout_number) subclass-specific time out method

void recv(Packet*, Handler*) receiving agent's main receive path

The *recv()* function is the main entry point for an agent which receives packets, and is invoked by upstream nodes when sending a packet. In most cases, agents make no use of the second argument (the handler defined by upstream nodes).

2.5.3 Protocol Agents

There are several agents supported in the simulator: The following are their class names:

TcpAgent a "Tahoe" TCP sender (cwnd = 1 on any loss)

TcpSink a Reno or Tahoe TCP receiver

CBR_Agent connectionless protocol with sequence numbers

UDP_Agent UDP with sequence numbers and traffic sources

LossMonitor a packet sink which checks for losses

NullAgent a degenerate agent which discards packets

The class *TcpAgent* represents a simplified TCP sender. It sends data to a *TcpSink* agent and processes its acknowledgments.

2.6 Schedulers and Events

There are three schedulers included in the simulator, each of which is implemented using a different data structure: a simple linked-list (default), heap, and calendar queue. The scheduler runs by selecting the next earliest event, executing it to completion, and returning to execute the next event. Currently the simulator is single-threaded, and only one event in execution at any given time. If more than one event is scheduled to execute at the same time, they are ordered (in some scheduler-dependent way) and executed serially. No partial execution of events or pre-emption is supported.

2.6.1 List Scheduler

The list scheduler implements the scheduler using a simple linked-list structure. The list is kept in time-order (earliest to latest), so event insertion and deletion require scanning the list to find the appropriate entry. Choosing the next event for execution requires trimming the first entry off the head of the list. This implementation preserves event execution in a FIFO manner for simultaneous events.

2.6.2 Heap Scheduler

The heap scheduler implements the scheduler using a heap structure. This structure is superior to the list structure for a large number of events, as insertion and deletion times are in O(log n) for n events.

2.6.3 Calendar Queue Scheduler

The calendar queue scheduler uses a data structure analogs to a one-year desk calendar, in which events on the same month/day of multiple years can be recorded in one day.

2.6.4 Event

An event generally comprises a "firing time" and a handler function.

```
class Event {
public:
        Event() : time_(0), uid_(0) {}
        ~Event();
        Event* next_;          /* event list */
        Handler* handler_;     /* handler to call when event ready */
        double time_;          /* time at which event is ready */
        int uid_;              /* unique ID */
};

/*
 * The base class for all event handlers. When an event's scheduled
 * time arrives, it is passed to handle which must consume it.
 */
class Handler {
public:
        virtual void handle(Event* event) = 0;

};
```

Two types of objects are derived from the base class *Event*: packets and "at-events". Packets are the fundamental unit of exchange between objects in the simulation. An at-event is a function execution scheduled to occur at a particular time.

Events are scheduled using the *at Simulator* function that allows functions to be invoked at arbitrary points in simulation time. They can be used to start or stop sources, dump statistics, instantiate link failures, reconfigure the network topology etc. The simulation is started via the run functions and continues until there are no more events to be processed. Invoking the *halt Simulator* function can prematurely halt the simulation.

Packets are forwarded along the shortest path route from a source to a destination, where the distance metric is the sum of costs of the links traversed from the source to the destination. The cost of a link is one by default; the distance metric is simply the hop count in this case. The cost of a link can be changed with the *cost Simulator* function. A static topology model is used as the default in Ns2 in which the states of nodes/links do not change during the course of a simulation. Also static unicast routing is the default in which the routes are pre-computed over the entire topology once prior to starting the simulation.

2.7 Queue Management and Packet Scheduling

Queues represent locations where packets may be held (or dropped). Packet scheduling refers to the decision process used to choose which packets should be serviced or dropped. Buffer management refers to any particular discipline used to regulate the occupancy of a particular queue. In HyNS, support is included for drop-tail (FIFO) queuing, RED buffer management, CBQ (including a priority and round-robin scheduler), Fair Queuing, and Stochastic Fair Queuing (SFQ). In the common case where a delay element is downstream from a queue, the queue may be blocked until it is re-enabled by its downstream neighbor. This is the mechanism by which transmission delay is simulated. Packet drops are implemented in such a way that queues contain a "drop destination"; that is, an object that receives all packets dropped by a queue. This can be useful to keep statistics on dropped packets.

2.7.1 Queue Class

The *Queue* class is derived from a *Connector* base class.

```
class Queue : public Connector {
public:
        virtual void enque(Packet*) = 0;
        virtual Packet* deque() = 0;
        void recv(Packet*, Handler*);
        void resume();
        int blocked() const { return (blocked_ == 1); }
        void unblock() { blocked_ = 0; }
        void block() { blocked_ = 1; }
        void reset();
        int qlim_;                  /* maximum allowed pkts in queue */

protected:
        Queue();
        int blocked_;               /* blocked now? */
        int unblock_on_resume_;     /* unblock q on idle? */
        QueueHandler qh_;
};
```

The *enque()* and *deque()* functions are pure virtual, indicating the *Queue* class is to be used as a base class; particular queues are derived from *Queue* and implement these two functions as necessary. In general, particular queues do not override the *recv()* function because it invokes the particular *enque()* and *deque()*.

The member variable *qlim_* is constructed to set a bound on the maximum queue length, but the Queue class itself does not enforce this; it must be used by the particular queue subclasses if they need this value. The Boolean member variable *blocked_* indicates whether the queue is able to send a packet immediately to its downstream neighbor. When a queue is blocked, it is able to queue packets but not to send them.

2.7.2 Queue Blocking

A queue may be either blocked or unblocked at any given time. Generally, a queue is blocked when a packet is in transit between it and its downstream neighbor. A blocked queue will remain blocked as long as its downstream link is busy and the queue has at

least one packet to send. A queue becomes unblocked only when its *resume()* function is invoked by a downstream neighbor, usually when no packets are queued.

2.7.3 PacketQueue Class

The *Queue* class may implement buffer management and scheduling but do not implement the low-level operations on a particular queue. The *PacketQueue* class is defined for this purpose as follows:

```
class PacketQueue : public TclObject {
public:
        PacketQueue() : head_(0), tail_(&head_), len_(0) {}
        inline int length() const { return (len_); }
        virtual void enque(Packet* p);
        virtual Packet* deque();
        Packet* lookup(int n);
        /* remove a specific packet, which must be in the queue */
        virtual void remove(Packet*);
        /* Remove a packet, located after a given packet. Either could be 0. */
        void remove(Packet *, Packet *);
protected:
        Packet* head_;
        Packet** tail_;
        int len_;               // packet count
};
```

This class maintains a linked-list of packets. Particular scheduling or buffer management schemes may make use of several *PacketQueue* objects. The *enque()* function places the specified packet at the end of the queue and updates the len_ member variable. The *deque()* function returns the packet at the head of the queue and removes it from the queue and updates the counter *len_*, or returns NULL if the queue is empty. The *lookup()* function returns the nth packet from the head of the queue. The *remove()* function deletes

the packet stored in the given address from the queue and updates the counter. It causes an abnormal program termination if the packet does not exist.

2.7.4 DropTail Class

The *DropTail* class implements FIFO scheduling and drop-on-overflow buffer management that is most typically used in present-day Internet routers.

```
/* A bounded, drop-tail queue */
class DropTail : public Queue {
 public:
        DropTail() { q_ = new PacketQueue; }
        ~DropTail();

 protected:
        void enque(Packet*);
        Packet* deque();
        PacketQueue *q_;        /* underlying FIFO queue */
};
```

The base class *Queue* provides most of the needed functionality. The drop-tail queue maintains exactly one FIFO queue, implemented by including a *PacketQueue* object. Drop-tail implements its own versions of *enque()* and *deque()* as follows:

```
void DropTail::enque(Packet* p)
{
        q_->enque(p);
        if (q_->length() >= qlim_) {
                q_->remove(p);
                drop(p);
        }
}
Packet* DropTail::deque()
{
        return q_->deque();
}
```

Here, the *enque()* function first stores the packet in the internal packet queue (which has no size restrictions), and then checks the size of the packet queue versus qlim_. Drop-on-overflow is implemented by dropping the packet most recently added to the packet queue

if the limit is reached or exceeded. Simple FIFO scheduling is implemented in the *deque()* function by always returning the first packet in the packet queue.

2.8 Delays and Links

Delays represent the time required for a packet to traverse a link. The amount of time required for a packet to traverse a link is defined to be s/b + d where s is the packet size (as recorded in its IP header), b is the speed of the link in bits/sec, and d is the link delay in seconds. The implementation of link delays is closely associated with the blocking procedures described for queues in section 2.7.2

2.8.1 LinkDelay Class

The class *LinkDelay* is derived from the base class *Connector*. It is briefly excerpted below, only considering "non-dynamic" links:

```
class LinkDelay : public Connector {
 public:
        LinkDelay();
        ~LinkDelay();
        void recv(Packet* p, Handler*);
        void send(Packet* p, Handler*);
        void handle(Event* e);
        inline double delay() { return delay_; }
        inline double txtime(Packet* p) {
                hdr_cmn *hdr = (hdr_cmn*)p->access(off_cmn_);
                return (hdr->size() * 8. / bandwidth_);
        }
        inline double bandwidth() const { return bandwidth_; }
       void pktintran(int src, int group);

      double bandwidth_;    /* bandwidth of underlying link (bits/sec) */
     double delay_;         /* line latency */

protected:
        void reset();

        Event intr_;
```

```
        Event inTransit_;

private:
        void schedule_next();
};
```

The *recv()* function overrides the base class *Connector* version. Its is excerpted below, only considering "non-dynamic" links:

```
void LinkDelay::recv(Packet* p, Handler* h)
{
        double txt = txtime(p);
        Scheduler& s = Scheduler::instance();
        s.schedule(target_, p, txt + delay_);

        s.schedule(h, &intr_, txt);
}
```

For "non-dynamic" links, this method operates by receiving a packet, p, and scheduling two events. Assume these two events are called E1 and E2, and that event E1 is scheduled to occur before E2. E1 is scheduled to occur when the upstream node attached to this delay element has completed sending the current packet (takes time equal to the packet size divided by the link bandwidth). E1 is usually associated with a *Queue* object, and will cause it possibly to become unblocked. E2 represents the packet arrival event at the downstream neighbor of the delay element. Event E2 occurs later than E1, the time difference between E1 and E2 is the link delay in seconds.

2.9 Packet Headers and Formats

Objects in the class *Packet* are the fundamental unit of exchange between objects in the simulation. Packet headers are defined on a per-protocol basis, new protocols may define their own packet headers or may extend existing headers with additional fields. New

packet headers can be introduced into the simulator by defining a structure with the needed fields, and then modifying some of the simulator initialization code to assign a byte offset in each packet where the new header is to be located relative to others.

2.9.1 Packet Class

The class *Packet* is a subclass of *Event*, so that packet can be scheduled like event (e.g. for later arrival at some queue). The class *Packet* defines the structure of a packet and provides member functions to handle a free list of objects of this type. It is defined as follows:

```
class Packet : public Event {
public:
        Packet* next_;          // for queues and the free list
        static int hdrlen_;
        Packet() : bits_(0), datalen_(0), next_(0) { }
        unsigned char* const bits() { return (bits_); }
       Packet* copy() const;
       static Packet* alloc();
       static Packet* alloc(int);
       inline void allocdata(int);
      static void free(Packet*);
     inline unsigned char* access(int off) {
       if (off < 0)
              abort();
       return (&bits_[off]); }
    inline unsigned char* accessdata() {return data_;}
protected:
        static Packet* free_;
private:
        unsigned char* bits_;
        unsigned char* data_;          // variable size buffer for 'data'
        unsigned int datalen_; // length of variable size buffer
};
```

This class holds a pointer to a generic array of unsigned characters (commonly called the "bag of bits" or BOB for short) where packet header fields are stored. It also holds a pointer to packet "data" (which is often not used in simulations). The member variable

*bits_* contains the address of the first byte of the BOB that is implemented as a concatenation of all the structures defined for each packet header. Packet header structures have the names like hdr_xxx by convention, where xxx represents the specific header names. BOB generally remains a fixed size throughout a simulation, and the size is recorded in the *Packet::hdrlen_* member variable.

The other functions of the class *Packet* are for creating new packets and storing old (unused) ones on a private free list. The *alloc()* is a support function commonly used to create new packets. It is called by *Agent::allocpkt()* on behalf of agents and is thus not normally invoked directly by most objects. It first attempts to locate an old packet on the free list and if this fails allocates a new one using the *new* operator. Note that *Packet* class objects and BOBs are allocated separately. The *free()* method frees a packet by returning it to the free list. Packets are never returned to the system's memory allocator. Instead, they are stored in a free list when *Packet::free()* is called. The *copy()* function creates a new, identical copy of a packet with the exception of the *uid_* field, which is unique.

2.9.2 Common Header

Each packet in the simulator has a "common" header that is defined as follows:

```
struct hdr_cmn {
        int     ptype_;         // packet type
        int     size_;                  // simulated packet size
        int     uid_;                   // unique id
        int     error_;         // error flag
        double ts_;                     // timestamp: for q-delay measurement
        int     iface_;                 // receiving interface (label)

        static int offset_;             // offset for this header
        inline int& offset() { return offset_; }

        /* per-field member functions */
```

```
        inline int& ptype() { return (ptype_); }
        inline int& size() { return (size_); }
        inline int& uid() { return (uid_); }
        inline int& error() { return error_; }
        inline double& timestamp() { return (ts_); }
        inline int& iface() { return (iface_); }
};
```

This structure primarily defines fields used for tracing the flow of packets or measuring other quantities. The time stamp field is used to measure queuing delay at switch nodes. The *ptype_* field is used to identify the type of packets, which makes reading traces simpler. The *scheduler employs the uid_ field* in scheduling packet arrivals. The *size_* field is of general use and gives the simulated packet's size. Note that the actual number of bytes consumed in the simulation may not relate to the value of this field. Rather, it is used most often in computing the time required for a packet to be delivered along a network link. The *simulator uses the iface_ field* when performing multicast distribution tree computations. It is a label indicating on which link a packet was received.

2.9.3 Packet Header Manager

An object of the class *PacketHeaderManager* is to manage the set of currently active packet header types and assign each of them unique offsets in the BOB. It is defined as follows:

```
class PacketHeaderManager {
public:
        PacketHeaderManager();
        static int vartab_[256];
        int allochdr(int );

};

#defineNS_ALIGN    8        /* byte alignment for structs */
int PacketHeaderManager::allochdr(int hdrlen)
{
```

```
        int size = hdrlen;

        // round up to nearest NS_ALIGN bytes
        int incr = (size + (NS_ALIGN-1)) & ~(NS_ALIGN-1);
        int base = Packet::hdrlen_;
        Packet::hdrlen_ =Packet::hdrlen_ + incr;

        return(base);
}

// Packet offsets
#define OFF_COMMON          0
#define OFF_IP              1
#define OFF_TCP             2
#define OFF_FLAGS           3
#define OFF_RTP             4
#define OFF_CELL            5   // ATM cell header

// set up the packet format for the simulation
void Simulator::create_packetformat()
{
        PacketHeaderManager* pm = new PacketHeaderManager;
        PacketHeaderManager::vartab_[OFF_COMMON] =
                                        pm->allochdr(sizeof(hdr_cmn));
        PacketHeaderManager::vartab_[OFF_IP] =
                                        pm->allochdr(sizeof(hdr_ip));
        PacketHeaderManager::vartab_[OFF_TCP] =
                                        pm->allochdr(sizeof(hdr_tcp));
        PacketHeaderManager::vartab_[OFF_FLAGS] =
                                        pm->allochdr(sizeof(hdr_flags));
        PacketHeaderManager::vartab_[OFF_RTP] =
                                        pm->allochdr(sizeof(hdr_rtp));
        PacketHeaderManager::vartab_[OFF_CELL] =
                                        pm->allochdr(sizeof(hdr_cell));

        packetManager_ = pm;

}
```

Function *Simulator::create_packetformat()* is called one time during simulator configuration. It first creates a single *PacketHeaderManager* object. After creating the packet manager, it enables each of the packet headers of interest. The placement of

headers is performed by the *allochdr()* function of the *PacketHeaderManager* class. The function keeps a running variable *Packet::hdrlen_* with the current length of BOB as new packet headers are enabled. It also arranges for 8-byte alignment for any newly enabled packet header. This is needed to ensure that when double-world length quantities are used in packet headers on machines where double-word alignment is required, access faults are not produced.

2.10 Traffic Generation

There are two methods of traffic generation in HyNS. One method uses the abstract class *TrafficGenerator* to generate inter-packet intervals and packet sizes. Classes derived from *TrafficGenerator* are used in conjunction with the *UDP_Agent* objects, which are responsible for actually allocating and transmitting the generated packets. The second method of traffic generation uses the *Source* class. Source objects generate traffic that is transported by *TcpAgent* objects.

2.10.1 Traffic Generator Class

TrafficGenerator is an abstract class defined as follows:

```
class TrafficGenerator : public NsObject {
public:
        TrafficGenerator() {}
        virtual double next_interval(int &) = 0;
        virtual void init() {}
protected:

        void recv(Packet*, Handler*) { abort(); }
        int size_;
};
```

The pure virtual function *next_interval()* returns the time until the next packet is created and also sets the size in bytes of the next packet. The member function *init()* is called by the *UDP_Agent* with which the *TrafficGenerator* is associated when the agent is started. Necessary initializations specific to a traffic generation process are done in *init()*.

There are three classes derived from the class *TrafficGenerator*:

1. *EXPOO_Source* -- generates traffic according to an Exponential On/Off distribution. Packets are sent at a fixed rate during on periods, and no packets are sent during off periods. Both on and off periods are taken from an exponential distribution. Packets have constant size.
2. *POO_Source* -- generates traffic according to a Pareto On/Off distribution. This is identical to the Exponential On/Off distribution, except the on and off periods are taken from a pareto distribution. These sources can be used to generate aggregate traffic that exhibits long range dependency.
3. TrafficTrace -- generates traffic according to a trace file. Each record in the trace file consists of 2 32-bit fields. The first contains the time in microseconds until the next packet is generated. The second contains the length in bytes of the next packet.

2.10.2 UDP Agent Class

TrafficGenerator objects merely generate inter-packet times and sizes. They do not actually allocate packets, fill in header fields and transmit packets. Hence, each *TrafficGenerator* object must be associated with another object that performs these functions. This functionality is implemented in class *UDP_Agent* that is defined as follows:

```
class UDP_Agent : public CBR_Agent {
public:
```

```
        UDP_Agent();
        virtual void timeout(int);

    protected:
        void start();
        void stop();
        TrafficGenerator *trafgen_;
        virtual void sendpkt();
};
```

This class is derived from the class *CBR_Agent*. It differs only in the manner in which inter-packet times and packet sizes are determined. Whereas the *CBR_Agent* uses fixed size packets and constant inter-arrival times (with optional randomization added to the inter-packet times), *UDP_Agent* objects invoke the *next_interval()* function on an associated *TrafficGenerator* object to determine these values.

2.10.3 Source Class

Classes derived from the *Source* class are used to generate traffic for TCP. There are two classes derived from Source: *FTPSource* and *TelnetSource*. These classes work by advancing the count of packets available to be sent by a TcpAgent. The actual transmission of available packets is still controlled by TCP's flow control algorithm.

2.11 Trace and Monitoring Support

There are a number of ways of collecting output or trace data on a simulation. Generally, trace data are either displayed directly during execution of the simulation, or more commonly stored in a file to be post-processed and analyzed. There are two distinct types of monitoring capabilities supported by the simulator – traces and monitors. Trace object records each individual packet as it arrives, departs, or is dropped at a link or queue.

Monitor records counts of various interesting quantities such as packet and byte arrivals, departures.

A common header (*hdr_cmn*) is included in each packet in order to support traces. This common header includes a unique identifier for each packet, a packet type field (set by agents when they generate packets), and a packet size field (in bytes, used to determine the transmission time for packets).

Monitors are supported by a separate set of objects that are created and inserted into the network topology around queues. They provide a place where arrival statistics and times are gathered, and statistics are computed over time intervals.

2.11.1 Trace Support

Objects of the following classes are inserted directly in-line in the network topology:

EnqueTrace	a packet arrival (usually at a queue)
DequeTrace	a packet departure (usually at a queue)
DropTrace	packet drop (packet delivered to drop-target)
SnoopQueueIn	on input, collect a time/size sample (pass packet on)
SnoopQueueOut	on output, collect a time/size sample (pass packet on)
SnoopQueueDrop	on drop, collect a time/size sample (pass packet on)

Objects of *QueueMonitor* class are added in the simulation and referenced by the objects listed above. They are used to aggregate statistics collected by the *SnoopQueue* objects.

The following *Simulator* member functions can be used to attach trace elements:

- *void flush_trace() This* function flushes buffers for all trace objects in the simulation.

- *Trace* create_trace(int type, FILE* file, Node* src, Node* dst) This* function creates a new trace object of type *type* between the given *src* and *dst* nodes and attaches a file handle to it. The function returns the handle to the newly created trace object.
- *void trace_queue(Node* n1, Node* n2, FILE* file* This function enables *Enque*, *Deque*, and *Drop* tracing on the link between nodes n1 and n2.
- *QueueMonitor* monitor_queue(Node* n1, Node* n2, FILE* qtrace, double sampleInterval=0.1)* This function calls the *init_monitor()* function on the link between nodes *n1* and *n2*, it arranges for the creation of *SnoopQueue* and *QueueMonitor* objects which can be used to ascertain time-aggregated queue statistics.
- *void drop_trace(Node* n1, Node* n2, Trace* trace)* This function makes the given trace object the drop-target of the queue associated with the link between nodes *n1* and *n2*.

The *Simulator* member functions described above require the *trace()* and *init_monitor()* *SimpleLink* member functions. The *trace()* function is defined as follows:

```
// Build trace objects for this link and update the object linkage
void SimpleLink::trace(Simulator* ns, FILE* f)
{
        enqT_ = ns->create_trace(TRACE_ENQUE, f, fromNode_, toNode_);
        deqT_ = ns->create_trace(TRACE_DEQUE, f, fromNode_, toNode_);
        drpT_ = ns->create_trace(TRACE_DROP, f, fromNode_, toNode_);

        // insert drpT_ in between drophead_ and its target_
        NsObject* nxt = drophead_->target_;
        drophead_->target_ = drpT_;
        drpT_->target_ = nxt;

        queue_->drop_ = drophead_;
```

```
        // insert deqT_ in between queue_ and its target_
        deqT_->target_ = queue_->target_;
        queue_->target_ = deqT_;

        // insert enqT_ in between head_ and its target_
        enqT_->target_ = head_;
        head_ = enqT_;
}
```

This function establishes *Enque*, *Deque*, and *Drop* traces in the simulator Ns2 and directs their output to file handle f. The function assumes a queue has been associated with the link. It operates by first creating three new trace objects and inserting the *Enque* object before the queue, the *Deque* object after the queue, and the *Drop* object between the queue and its previous drop target. Note that all trace output is directed to the same file handle.

The following functions, *init_monitor()* and *attach_monitor()*, are used to create a set of objects used to monitor queue sizes of a queue associated with a link. They are defined as follows:

```
// like init_monitor, but allows for specification of more of the items
void SimpleLink::attach_monitors(SnoopQueue* insnoop, SnoopQueue*
outsnoop, SnoopQueue* dropsnoop, QueueMonitor* qmon)
{
        snoopIn_ = insnoop;
        snoopOut_ = outsnoop;
        snoopDrop_ = dropsnoop;

        snoopIn_->target_ = head_;
        head_ = snoopIn_;

        snoopOut_->target_ = queue_->target_;
        queue_->target_ = snoopOut_;

        NsObject* nxt = drophead_->target_;
        drophead_->target_ = snoopDrop_;
```

```
        snoopDrop_->target_ = nxt;

        snoopIn_->qm_ = qmon;
        snoopOut_->qm_ = qmon;
        snoopDrop_->qm_ = qmon;
        qMonitor_ = qmon;
}

// Insert objects that allow us to monitor the queue size
// of this link. Return the name of the object that
// can be queried to determine the average queue size.
QueueMonitor* SimpleLink::init_monitor(Simulator* ns, FILE* qtrace, double
sampleInterval)
{
        ns_ = ns;
        qtrace_ = qtrace;
        sampleInterval_ = sampleInterval;
        qMonitor_ = new QueueMonitor;

        SnoopQueueIn* snoopIn = new SnoopQueueIn;
        SnoopQueueOut* snoopOut = new SnoopQueueOut;
        SnoopQueueDrop* snoopDrop = new SnoopQueueDrop;
        attach_monitors(snoopIn, snoopOut, snoopDrop, qMonitor_);

        Integrator* bytesInt = new Integrator;
        qMonitor_->bytesInt_ = bytesInt;

        Integrator* pktsInt = new Integrator;
        qMonitor_->pktsInt_ = pktsInt;

        return (qMonitor_);
}
```

These functions establish queue monitoring on the *SimpleLink* object in the simulator. Queue monitoring is implemented by constructing three *SnoopQueue* objects and one *QueueMonitor* object. The *SnoopQueue* objects are linked in around a *Queue* in a way similar to *Trace* objects. The *SnoopQueueIn(Out)* object monitors packet arrivals(departures) and reports them to an associated *QueueMonitor* agent. In addition, a *SnoopQueueOut* object is also used to accumulate packet drop statistics to an associated

QueueMonitor object. The same *QueueMonitor* object is used in all cases for *init_monitor()*.

2.11.2 Trace Class

The *Trace* class is defined as follows:

```
class Trace : public Connector {
 public:
        Trace(int type);
        ~Trace();

        inline void detach() { channel_ = NULL; }
        inline void flush() { fflush(channel_); }

        inline void attach(FILE* fileHandle){ channel_ = fileHandle;}

        void recv(Packet* p, Handler*);
        virtual void trace(TracedVar*);
        void dump();
        inline char* buffer() { return (wrk_); }

        int off_ip_;
        int off_tcp_;
        int off_rtp_;
        int off_cell_;

        nsaddr_t src_;
        nsaddr_t dst_;

 protected:
        int type_;

        FILE* channel_;
         int callback_;
        char wrk_[256];
        void format(int tt, int s, int d, Packet* p);
         void annotate(const char* s);
        int show_tcphdr_;           // bool flags; backward compat
};
```

The *src_*, and *dst_* internal state is used to label trace output and is independent of the corresponding field names in packet headers. The main *recv()* function is defined as follows:

```
void Trace::recv(Packet* p, Handler* h)
{
        format(type_, src_, dst_, p);
        dump();
        if (target_ == 0)
                Packet::free(p);
        else
                send(p, h);
}
```

The function merely formats a trace entry using the source, destination, and particular trace type character. The dump function writes the formatted entry out to the file handle *channel_*.

2.11.3 Trace File Format

The *Trace::format()* function sets the trace file format, it is defined as follows:

```
void Trace::format(int tt, int s, int d, Packet* p)
{
        hdr_cmn *th = (hdr_cmn*)p->access(off_cmn_);
        hdr_ip *iph = (hdr_ip*)p->access(off_ip_);
        hdr_tcp *tcph = (hdr_tcp*)p->access(off_tcp_);
        hdr_rtp *rh = (hdr_rtp*)p->access(off_rtp_);
        hdr_cell *cellh = (hdr_cell*)p->access(off_cell_);
        int t = th->ptype();
        const char* name = pt_names[t];

        if (name == 0)
                abort();

        int seqno;
        if (t == PT_RTP || t == PT_CBR)
                seqno = rh->seqno();
        else if (t == PT_TCP || t == PT_ACK)
                seqno = tcph->seqno();
        else if (t == PT_CELL )
```

```
            seqno = cellh->seq();
        else
            seqno = -1;

        ...

        if (!show_tcphdr_) {
            sprintf(wrk_,
                    "%c %g %d %d %s %d %s %d %d.%d %d.%d %d %d",
                    tt,
                    Scheduler::instance().clock(),
                    s,
                    d,
                    name,
                    th->size(),
                    flags,
                    iph->flowid(),
                    iph->src() >> 8, iph->src() & 0xff,
                    iph->dst() >> 8, iph->dst() & 0xff,
                    seqno,
                    th->uid());
        } else {
            sprintf(wrk_,
              "%c %g %d %d %s %d %s %d %d.%d %d.%d %d %d %d 0x%x %d",
                    tt,
                    Scheduler::instance().clock(),
                    s,
                    d,
                    name,
                    th->size(),
                    flags,
                    iph->flowid() ,
                    iph->src() >> 8, iph->src() & 0xff,
                    iph->dst() >> 8, iph->dst() & 0xff,
                    seqno,
                    th->uid(),
                    tcph->ackno(),
                    tcph->flags(),
                    tcph->hlen());
        }
    }
```

This function formats the source, destination, type fields defined in the trace object, the current time, along with various packet header fields including: type of packet (as a

name), size, flags (symbolically), flow identifier, source and destination packet header fields, sequence number (if present), and unique identifier. The *show_tcphdr_* variable indicates whether the trace output should append tcp header information (ack number, flags, and header length) at the end of each output line.

2.12 Static Routing and Route Logic Class

HyNS applies the static routing strategy that runs the same routing protocol on all the nodes in the topology. The route computation algorithm is run exactly once prior to the start of the simulation. The routes are computed using an adjacency matrix and link costs of all the links in the topology. The main class of route logic is *RouteLogic*, the routing architecture is also implemented in the following classes: *Simulator*, *Link*, *Node* and *Classifier*.

The class *RouteLogic* is defined as follows:

```
/*
 * Routing codes for general topologies based on min-cost routing algorithm in
 * Bertsekas' book. Written originally by S. Keshav, 7/18/88
 */

class RouteLogic : public TclObject {
public:
        RouteLogic();
        ~RouteLogic();
        inline void compute() {compute_routes();}
        int lookup(int src, int dst);

        void insert(int src, int dst, int cost);
        void configure();
        void print_route();

protected:
        void check(int);
        void alloc(int n);
```

```
        void reset(int src, int dst);
        void compute_routes();
        int* adj_;
        int* route_;
        int size_;
        int maxnode_;

};
void RouteLogic::configure()
{
   // Since the Simulator knows the entire topology, its member function
   // compute_routes() is well suited here.
   Simulator::instance().compute_routes();
}
```

The *configure()* function is invoked by the Simulator *run()* function to setup routes before simulation starts.

The *lookup()* function takes source node ID (src), destination node ID (dst), and returns the ID of the neighbor node that src uses to reach dst. This function is used by the static route computation function to query the computed routes and populate the routes at each of the nodes.

3. OTcl/C++ Conversion

Ns2 is an object-oriented simulator, written in C++, with an OTcl interpreter as a front-end. The simulator supports a class hierarchy in C++ (the compiled hierarchy), and a similar class hierarchy within the OTcl interpreter (the interpreted hierarchy). The two hierarchies are closely related to each other; from the user's perspective, there is a one-to-one correspondence between a class in the interpreted hierarchy and one in the compiled hierarchy. The control operations of Ns2 are in OTcl and data are passed through C++ objects for speed.

3.1 Why Take Out OTcl Part

We can see from above that Ns2 supports two similar class hierarchies in C++ and OTcl, and this overhead slows down the simulation. Also, any one wishing to fully understand or modify it must be comfortable with both environments and tools. Plus, it is painful to do OTcl and C++ debugging together. Imagine, when looking at the OTcl code and debugging OTcl level stuff, one wants to look at the C++-level classes, and vice versa. Since not many researchers are familiar with OTcl, we decided to take out the OTcl part and convert OTcl code into C++ code, this way, user only needs to know C++ to understand the simulator and can step into each line of code by using only C debugger.

Ns2 is a fairly big simulator; it has 27K lines of C++ code and 12K lines of OTcl support code. In order to finish this project within certain time limit, the simulator is simplified by removing some detailed or complex implementation, hence, not all the OTcl code is translated and not all the C++ classes are included in HyNS.

Since Ns2 uses very fine-grain objects, as long as the original class structure is maintained, it is very easy to add back all the original features when needed, and can evolve along with Ns2.

3.2 What was Done to Remove OTcl Part

The main simulation classes of Ns2 like *Simulator*, *Node* and *Link* are written fully in OTcl, and some classes are defined in both C++ and OTcl, like *Source*, *TelnetSource*, *PacketHeaderManager*, *Agent*, *Trace*, *QueueMonitor* and *RandomVariable*. Other OTcl classes are *FTPSource*, *EnqueTrace*, *DequeTrace* and *DropTrace*. All these OTcl classes or OTcl functions need to be converted into C++ classes and member functions. Since Ns2 supports two similar class hierarchies in C++ and OTcl, the OTcl linkage need to be removed from C++ code and the class hierarchy within the OTcl interpreter needs to be removed.

Following are the tasks done to remove the OTcl part:

- Translation of OTcl Code
- Removal of OTcl Linkage
- Removal of the interpreted class hierarchy

3.3 Translation of OTcl Code

The three main OTcl classes (*Simulator*, *Node* and *Link)* and other OTcl classes (FTPSource, *EnqueTrace*, *DequeTrace* and *DropTrace*) are translated into C++ classes. The OTcl functions of the mixed classes (Source, *TelnetSource*, *PacketHeaderManager*, *Agent*, *Trace*, *QueueMonitor* and *RandomVariable*) are translated into C++ member functions. Some of the OTcl code is modified before been translated into C++ code.

3.3.1 Comparison with C++

In order to translate OTcl to C++ code correctly, we need to know the differences between OTcl and C++. Some of the differences are as follows [OTcl95]:

- While OTcl has multiple definitions, C++ has single class declaration. Each method definition (with instproc) adds a method to a class. Each instance variable definition (with set or via instvar in a method body) adds an instance variable to an object.
- While C++ has constructor/destructor, OTcl has init/destroy methods. Unlike constructors and destructors, init and destroy methods do not combine with base classes automatically. They should be combined explicitly with next.
- Unlike C++, OTcl methods are always virtual and called through the object.
- In OTcl, the name ***self***, which is equivalent to ***this*** in C++, may be used inside method bodies.
- C++ shadowed methods are called explicitly with scope operator, while OTcl calls them with ***next***. ***next*** searches further up the inheritance graph to find shadowed methods automatically. It allows methods to be combined without naming dependencies.
- Unlike C++, OTcl has no static methods and variables.
- Unlike C++, OTcl is a typeless, string-oriented language. C++ arrays use integer-valued index while OTcl arrays use string-valued index. So, an OTcl array is actually a mapping from string to string.

3.3.2 Translation Examples

Translating the OTcl code to C++ is harder than it sounds, partly because transformation from OTcl to C++ is not all that straightforward. Since OTcl has no data type, all the variable data types have to be figured out from the context.

The core OTcl class ***Simulator*** is the principal interface to the simulation engine; the following is the constructor of this class in OTcl:

```
Simulator instproc init args {
        eval $self next $args
        $self create_packetformat
        $self instvar scheduler_ nullAgent_
        set scheduler_ [new Scheduler/List]
        set nullAgent_ [new Agent/Null]
}
```

instproc is used for method definition, it defines ***init*** method for the class ***Simulator***. ***init*** method corresponds to the constructor function in C++, ***self*** is equivalent to ***this*** in C++, ***set*** or ***instvar*** is used for instance variable definition, it adds instance variables to the ***Simulator*** object.

The following is the translation of class ***Simulator*** constructor function in C++, all the member variables and member functions are declared once in the header file, see section 3.3.3.2.

```
Simulator::Simulator()
{
        create_packetformat();
        scheduler_ = (Scheduler*) new ListScheduler;
        nullAgent_ = new NullAgent(0);
        routingTable_ = NULL;
        atm_routingTable_ = NULL;
}
```

Since OTcl arrays use string-valued index, an OTcl array can be translated into a one, two or more dimensional array according to the constitution of the string-valued index. The OTcl array ***link_($sid:$did)*** stores the link object from source (with node id equals to

sid) to destination (with node id equals to *did*). The index string is the concatenation of source node ID, separator ': ', and destination node ID. This array can be translated into two dimensional C array ***link_[sid][did]***.

The following is the OTcl code sample where the array ***link_($sid:$did)*** is in use:

```
Simulator instproc simplex-link { n1 n2 bw delay type } {
        $self instvar link_ nullAgent_
        $self instvar traceAllFile_

        set sid [$n1 id]
        set did [$n2 id]

        set q [new Queue/$type]
        $q drop-target $nullAgent_

        set link_($sid:$did) [new SimpleLink $nd1 $nd2 $bw $delay $q]
        $n1 add-neighbor $n2

        if [info exists traceAllFile_] {
                $self trace-queue $n1 $n2 $traceAllFile_
        }
}
```

The C++ translation of the above OTcl code is shown as following:

```
void Simulator::simplex_link(Node* n1, Node* n2, double bw, double delay, int
type)
{
        int sid = n1->id();
        int did = n2->id();

        Queue* q;
        if (type == DROP_TAIL)
                q = new DropTail;
        else // (type == RED_QUEUE)
                q = new REDQueue;
        q->drop_ = nullAgent_;

        link_[sid][did] = new SimpleLink(sid, did, bw, delay, q);

        n1->add_neighbor(n2);

        if (traceAllFile_)
```

```
        trace_queue(n1, n2, traceAllFile_);
}
```

3.3.3 Translated Main Simulation Classes

The class hierarchy of HyNS is shown in Figure 3. Some of the classes are explained in this section and others will be explained in later sections where they are mentioned.

3.3.3.1 Root Class TclObject

Most of the classes in the HyNS are derived from a single base class - *TclObject* - at the root of the class hierarchy. The class name *TclObject* is inherited from Ns2, but all the Tcl related functions are removed, only naming capability is kept and enhanced. By abstracting the naming functions to the root class, all the subclass objects can have names associated with them. Object names are very useful when it comes to simulation debugging and printing, since names are much more meaningful than addresses. *TclObject* class is defined as follows:

```
class TclObject {
public:
        virtual ~TclObject();

        inline const char* name() { return (name_); }  // retrieve name
        void name(const char* s);                      // set name to s
        void name(const char* s, int id);              // set name to s+id
        void name(const char* s, int id1, int id2);    // set name to s+id1+id2

protected:
        TclObject();
        char* name_;
};
```

HyNS Class Chart

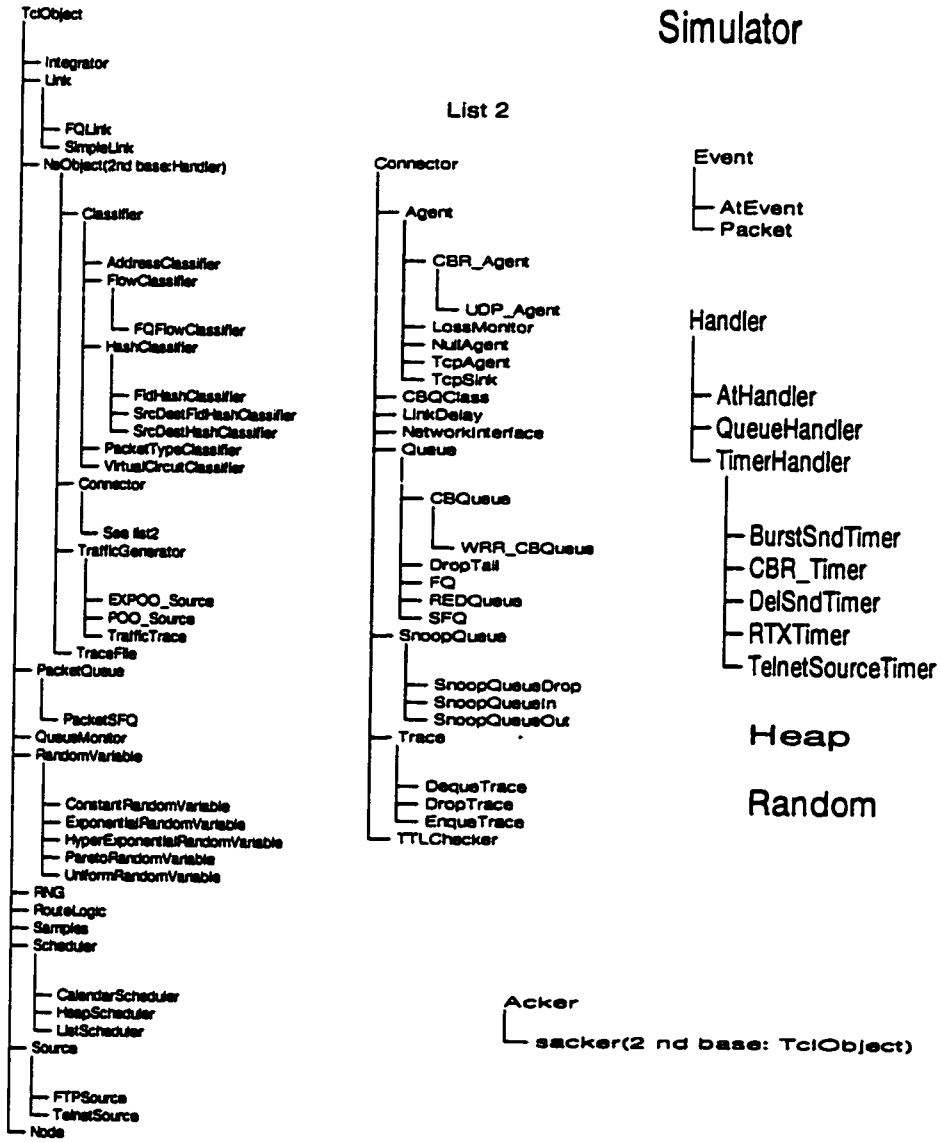

Figure 3 HyNS Class Hierarchy

3.3.3.2 Class Simulator

Class *Simulator* describes the overall simulator. It provides a set of functions for configuring a simulation and for choosing the type of event scheduler used to drive the simulation. The configuration generally begins by creating an instance of *Simulator* class and calling various functions to create and manage the topology, and internally stores references to each element of the topology. The class is defined as follows:

```
class Simulator {
 public:
        Simulator();
        ~Simulator();

        static Simulator& instance() { return (*instance_); }
        static Agent* nullAgent_;
        static Node* Node_[MAX_NODES];   // array of references to the nodes,
                                         // Indexed by the node id
        static Link* link_[][MAX_NODES];
        static int EnableMcast_;
        static int nt_;                  // total number of trace in alltrace_[]
        static int nc_;                  // number of TCP connections
        Scheduler* scheduler_;
        Node* node(int);
        void now();
        void at(Event*);
        int at(double, void* (*proc)(void*), void* args=0);
        void cancel(Event*);
        void run();
        void halt();
        RouteLogic* get_routelogic();
        RouteLogic* get_atm_routelogic();
        void simplex_link(Node*, Node*, double, double, int);
        void duplex_link(Node*, Node*, double, double, int);
        void use_scheduler(int);
        void delay(Node*, Node*, double);
        Trace* create_trace(int, FILE*, Node*, Node*);
        void trace_queue(Node*, Node*, FILE*);
        void flush_trace();
        inline void trace_all(FILE* file) { traceAllFile_ = file;}
        QueueMonitor* monitor_queue(Node*, Node*, FILE*, double);
        void queue_limit(Node*, Node*, int);
        void drop_trace(Node*, Node*, Trace*);
        void cost(Node*, Node*, int);
```

```
        void attach_agent(Node*, Agent*);
        void detach_agent(Node*, Agent*);
        Agent* connect(Agent*, Agent*, Node** route=NULL, int cnt = 0);
        void compute_routes();
        void print_conn_table();

    protected:
        RouteLogic* routingTable_;
        RouteLogic* atm_routingTable_;

        FILE* traceAllFile_;
        Trace* alltrace_[MAX_NODES];          // array of trace
        static Simulator* instance_;
        PacketHeaderManager* packetManager_;
        int connection_[MAX_CONNECTIONS][4];  // global mapping table
        Node** route_list_[MAX_NODES];        // route list indexed by conn.
        void create_packetformat();
        inline int alloc_conn() { return (nc_++); }
        void create_vc(int, int, int);
        void conn_gateways();
};
```

3.3.3.3 Class Node

Each node has a unique node ID to uniquely identify each node in the topology. Passing an integer node type parameter to the Simulator function *node (int node_type)* creates IP nodes, ATM switches or gateways, and a unique ID is automatically assigned to each node. Nodes are composed of classifiers. The class is defined as follows:

```
class Node : public TclObject {
 public:
        Node(int);
        ~Node();
        static int nn_;                       // total number of nodes
        inline int id() { return (id_); }     // returns the node ID
        int type_;                            // node type
        inline Classifier* entry() { return (classifier_); }
        void add_neighbor(Node*);             // adds a neighbor to the neighbor list
        inline NodeList* neighbors(){ return (neighbor_); }
        void add_route(int dst, NsObject* target
        void add_switch(int vci, NsObject* target);
```

```
        Agent* agent(int port);
        inline int alloc_port() { return (np_++); }
        void attach(Agent*);              // add the agent to agent list agents_
        void detach(Agent*, Agent*);      // remove the agent from agents_
        void reset();                     // reset all agents at the node

protected:
        Classifier* classifier_;
        inline int getid() { return(nn_++); }
        int id_;                          // 0 based node ID
        NodeList* neighbor_;
        AgentList* agents_;
        AddressClassifier* dmux_;         // port classifier
        int np_;                          // number of local agents (each has a port)
};
```

3.3.3.4 Class Link and SimpleLink

The class *Link* provides a few simple primitives. The class *SimpleLink* is the subclass of class *Link* that provides the ability to connect two nodes with a point to point link with an associated queue and delay. As with the node being composed of classifiers, a simple link is built up from a sequence of connectors.

Class *Link* is defined as follows:

```
class Link : public TclObject{
public:
        Link(int, int);
        ~Link();

        inline NsObject* head() { return(head_); }
        virtual inline Queue* queue() { return(queue_); }
        inline LinkDelay* link() { return(link_); }

        inline void cost(int c) { cost_ = c; }        // set link cost to value c
        inline int get_cost() { return(cost_);}

        virtual void trace(Simulator* ns, FILE* f) {};

protected:
        NsObject* head_;        // Entry point to the link, it points
                                // to the first object in the link.
```

```
    Queue* queue_;          // Reference to the main queue element of the link
    LinkDelay* link_;       // A reference to the element that actually
                            // models the link, in terms of the delay and
                            // bandwidth characteristic of the link.
    Trace* trace_;
    Node* fromNode_;        // source node
    Node* toNode_;          // destination node
    int source_;            // source node id
    int dest_;              // destination node id
    int cost_;              // link cost

};
```

Class *SimpleLink* is derived from the base class *Link* as follows:

```
class SimpleLink : public Link {
public:
    SimpleLink(int, int, double, double, Queue*);
    ~SimpleLink();

    void trace(Simulator*, FILE*);          // Build trace objects for this link
                                            // and update the object linkage
    QueueMonitor* init_monitor(Simulator*, FILE*, double);

protected:
    void attach_monitors(SnoopQueue*, SnoopQueue*, SnoopQueue*,
QueueMonitor*);
    void start_tracing();
    void ttl_drop_trace(Connector*);
    void ttl_drop_trace();
    void transit_drop_trace();
    void transit_drop_trace(Connector*);
    void queue_sample_timeout();
    char* sample_queue_size();
    TTLChecker* ttl_;   // Reference to the element that manipulates the ttl
                        // in every packet.
    Connector* drophead_; // Reference to an object that is the head of
                          // queue of elements that process link drops.
    Trace* enqT_;       // Reference to the element that traces
                        // packets entering queue_
    Trace* deqT_;       // Reference to the element that traces
                        // packets leaving queue_
    Trace* drpT_;       // Reference to the element that traces
                        // packets dropped from queue_
    SnoopQueue* snoopIn_;
```

```
        SnoopQueue* snoopOut_;
        SnoopQueue* snoopDrop_;
        QueueMonitor* qMonitor_;

        Simulator* ns_;
        FILE* qtrace_;              // queue trace output file pointer
        double sampleInterval_;     // float number
        double lastSample_;         // last sample time
};
```

The constructor is defined as follows:

```
SimpleLink::SimpleLink(int src, int dst, double bw, double delay, Queue* q)
        : Link(src, dst)
{
        queue_ =q;
        link_ = new LinkDelay;
        link_->bandwidth_ = bw;
        link_->delay_ = delay;
        link_->name("LinkDelay", src, dst);

        queue_->target_ = link_;
        link_->target_ = (NsObject*)Simulator::Node_[dst]->entry();

        head_ = (NsObject*) queue_;

        drophead_ = new Connector;
        drophead_->target_ = (NsObject*)Simulator::nullAgent_;
        drophead_->name("DropHead", src, dst);

        ttl_ = new TTLChecker;
        ttl_->target_ = link_->target_;
        ttl_->name("TTLChecker", src, dst);
        ttl_drop_trace();
        link_->target_ = ttl_;

        lastSample_ = 0;
}
```

Notice that when a *SimpleLink* object is created new *LinkDelay* and *TTLChecker* objects are also created, and the *Queue* object must have already been created.

3.4 Removal of OTcl Linkage

Users create new simulator objects through the interpreter; these objects are instantiated within the interpreter, and are closely mirrored by a corresponding object in the compiled hierarchy. There are six OTcl classes used in Ns2:

- Class Tcl contains the methods that C++ code will use to access the interpreter.
- Class TclObject is the base class for all simulator objects that are also mirrored in the compiled hierarchy.
- Class TclClass defines the interpreted class hierarchy, and the methods to permit the user to instantiate TclObjects.
- Class TclCommand is used to define simple global interpreter commands.
- Class EmbeddedTcl contains the methods to load higher-level built-in commands that make configuring simulations easier.
- Class InstVar contains methods to access C++ member variables as OTcl instance variables.

3.4.1 Variable Bindings

In most cases, access to compiled member variables is restricted to compiled code, and access to interpreted member variables is likewise confined to access via interpreted code. However, it is possible to establish bi-directional bindings such that both the interpreted member variable and the compiled member variable access the same data, and changing the value of either variable changes the value of the corresponding paired variable to the same value. In Ns2, the compiled constructor establishes the binding when that object is instantiated; it is automatically accessible by the interpreted object as an instance variable.

3.4.2 command Methods

For every TclObject that is created, Ns2 establishes the instance procedure, cmd{}, as a hook to executing methods through the compiled shadow object. The procedure cmd{} invokes the method command() of the shadow object automatically, passing the arguments to cmd{} as an argument vector to the command() method.

3.4.3 Class TclClass

This compiled class TclClass is a pure virtual class. Classes derived from this base class provide two functions: construct the interpreted class hierarchy to mirror the compiled class hierarchy; and provide methods to instantiate new TclObjects. Each such derived class is associated with a particular compiled class in the compiled class hierarchy, and can instantiate new objects in the associated class.

3.4.4 Breaking the OTcl Linkage

To remove the OTcl linkage from C++ code, following modifications are done to all the OTcl classes:

- Set up all the default parameters directly in C++ instead of variable bindings.
- Remove command() function and extract the useful functions if not already exists in C++ code, since all the functions will be called directly from C++ code.
- Remove all the subclasses of class TclClass, since there is no need for the interpreted class hierarchy.

4. IP over ATM Network Simulation

In order to support ATM and TCP/IP over ATM, following components are added to HyNS:

- TCP connection to ATM virtual circuit mapping
- Second layer routing (ATM switching)
- ATM switch (Virtual Circuit Classifier)
- IP/ATM gateway function (segmentation and reassemble)

In HyNS, the protocol stack of "TCP/IP/AAL5/ATM" is used to implement TCP/IP over ATM. Here, the functionality of AAL5 is implemented in IP/ATM gateways. An IP packet from the source computer goes into a router, then through an IP/ATM gateway that segments the packet into cells. The segmented cells go through the ATM network to an IP/ATM gateway near the other point, where they are reassembled back into the IP packet and forwarded to a router on the destination end, and finally into the destination computer.

4.1 TCP Connection to Virtual Circuit Mapping

All the mappings from TCP connection to virtual circuit are first set up and stored in member variable *Simulator::connection_*, then the related mapping information is extracted and stored in each gateway. The member variable *Simulator::nc_ keeps the total count of the TCP connections.* For fast mapping from both directions, each gateway

keeps two local mapping tables, one map from connection to virtual circuit (VC), the other maps from VC to connection.

4.1.1 Global Mapping Table

The global mapping table has four columns and the same number of rows as the total number of TCP connections. Each row stores the information about a TCP connection, TCP connection number is implied in the row number. The first column of the mapping table is the combination of source node ID and source agent port ID. The second column is the combination of destination node ID and destination agent port ID. The third column is the mapping virtual circuit ID, and the fourth column is the node count in the user specified path for the second layer switching. When a TCP connection is established between source and destination agents using function *Simulator::connect()*, a unique connection ID is created and a new row is added to the mapping table. The mapping virtual circuit ID is assigned later if the connection includes any ATM switch. The function *Simulator::connect()* is defined as follows:

```
Agent* Simulator::connect(Agent* src, Agent* dst, Node** route, int cnt)
{
        // Connect agents (low eight bits of addr are port number,
        // High 24 bits are node number)
        Node* srcNode = src->node_;
        Node* dstNode = dst->node_;

        int src_address = srcNode->id() << 8 | src->port();
        int dst_address = dstNode->id() << 8 | dst->port();

        // dst_ -- destination address for pkt flow
        src->dst_ = dst_address;
        dst->dst_ = src_address;

        // Populate global mapping table
        int conn = alloc_conn();              // get next connection number
        connection_[conn][CONN_SRC] = src_address;
```

```
        connection_[conn][CONN_DST] = dst_address;
        connection_[conn][CONN_VC] = -1; // no virtual circuit ID assigned yet
        connection_[conn][CONN_ROUTE_CNT] = cnt;
        route_list_[conn] = route;

        // Setup source agent's TCP connection number
        src->conn_ = conn;

        return(src);
}
```

Source agent's TCP connection number is also setup when the connection is established. This way, the source agent can include the connection number in the packet header when sending packets.

4.1.2 Local Mapping Tables

Each gateway keeps two local mapping tables (two one-dimensional arrays). One table maps from TCP connection to virtual circuit (*tcp2vc_*), indexed by connection number. The other table maps from virtual circuit to TCP connection (*vc2tcp_*), indexed by virtual circuit ID. After all TCP connections are established, function *Simulator::conn_gateways()* is called to route through all TCP connections. If gateway-gateway link is found and there is no physical link exists between these two gateways, a virtual circuit is created to connect these two gateways, the virtual circuit ID is assigned to the corresponding TCP connection in the global mapping table and the two local mapping tables. Function *Simulator::conn_gateways()* is defined as follows:

```
void Simulator::conn_gateways()
{
        int src_address, dst_address;
        int src_id, dst_id, new_src, found;
        RouteLogic* r = get_routelogic();
        int vcid = 0;       // virtual circuit id start from 0
        int nh;             // nexthop
        int i;
```

```
        for(i=0; i<nc_; i++) {
            src_address = connection_[i][CONN_SRC];
            dst_address = connection_[i][CONN_DST];

            // get node ID (lower eight bits of address are agent port ID,
            // higher 24 bits are node ID)
            src_id = (src_address >> AddressClassifier_shift_) &
                                            AddressClassifier_mask_;
            dst_id = (dst_address >> AddressClassifier_shift_) &
                                            AddressClassifier_mask_;

            // start from original source node, route through this connection to
            // find gateway-gateway virtual link and construct virtual circuit for it
            new_src = src_id;
            found = 0;
            PacketTypeClassifier* ptypeClassifier1;
            PacketTypeClassifier* ptypeClassifier2;

            nh = r->lookup(new_src, dst_id);  // find next hop
            while (nh >= 0 && nh != dst_id) {
                if (Node_[new_src]->type_ == GATEWAY &&
                    Node_[nh]->type_ == GATEWAY &&
                    link_[new_src][nh] == NULL) {
                            found = 1;
                            connection_[i][CONN_VC] = vcid;
                            ptypeClassifier1 =
                             (PacketTypeClassifier*)(Node_[new_src]->entry());
                            ptypeClassifier2 =
                            (PacketTypeClassifier*)(Node_[nh]->entry());
                            ptypeClassifier1->tcp2vc_[i] = vcid;
                            ptypeClassifier2->vc2tcp_[vcid] = i;
                            create_vc(i,new_src,nh);
                    }
                    new_src = nh;
                    nh = r->lookup(new_src, dst_id);
            }
            if (found == 1)
                    vcid++;
        }
    }
```

4.2 Two-Layered Routing

HyNS uses two-layered routing for IP over ATM networks, first layer is IP routing and second layer is ATM routing (switching). IP routing includes all the IP nodes (IP hosts and routers) and IP/ATM gateways, and all the gateways are fully meshed either by a real link or a virtual link. IP routing will always use the shortest path generated by the system. ATM switching includes all the gateways and ATM switches, connects pairs of gateways with switches along the shortest path generated by the system or along the user specified path.

4.2.1 Route Table Setup

Each simulator contains two *RouteLogic* pointers, one for IP routing (*routingTable_*) and the other for ATM switching (*atm_routingTable_*). Following function computes both route logic and populates route/switch table for each node:

```
void Simulator::compute_routes()
{
        // get both route logic
        RouteLogic* r = get_routelogic();              // for IP nodes and gateways
        RouteLogic* atm_r =  get_atm_routelogic(); // for switches and gateways

        // insert link cost for all the  real links in both route logic
        int i, j;
        Link* l;
        for(i=0; i<Node::nn_; i++) {
          for(j=0; j<Node::nn_; j++) {
                  l = link_[i][j];
                  if (l != NULL) {
                    // assume all the links are always up
                    if (Node_[i]->type_ != ATM_SWITCH &&
                      Node_[j]->type_ != ATM_SWITCH)
                              r->insert(i, j, l->get_cost());

                    if (Node_[i]->type_ != IP_NODE &&
                      Node_[j]->type_ != IP_NODE)
                              atm_r->insert(i, j, l->get_cost());
                  }
          }
```

```
}

// set up virtual links between all pairs of gateways
// in the first layer route logic if real links does not exist.
for(i=0; i<Node::nn_; i++) {
   for(j=0; j<Node::nn_; j++) {
      if (i != j && Node_[i]->type_ == GATEWAY &&
                    Node_[j]->type_ == GATEWAY) {
            if (link_[i][j] == NULL) // if not already exists, add it
               r->insert(i, j, VIRTUAL_LINK_COST);
            }
   }
}

// compute both route logic
r->compute();
atm_r->compute();

// set up route table for each first layer node (IP node or gateway)
i = 0;
Node* n1;
int nh;                    // next hop
Link* link;
while(i < Node::nn_ ) {
        n1 = Node_[i];
        if (n1->type_ != ATM_SWITCH) {
          int j = 0;
          while(j < Node::nn_) {
             if(i != j) {
                if (Node_[j]->type_ != ATM_SWITCH) {
                        nh = r->lookup(i, j);
                        if( nh >= 0 ) {
                                link = link_[i][nh];
                                if (link != NULL)
                                        n1->add_route(j, link->head());
                        }
                }
             }
             j++;
          }
        }
        i++;
}

// set up switch table for each second  layer node (switch or gateway)
conn_gateways();
```

}

A virtual link is set up when no real link exists between two gateways. The cost of this virtual link is set to be very high, higher than the cost of the longest real route (concatenation of all the real links along the worst route) in the topology. This way real route will always be chosen over virtual route (routes which include virtual links) based on the shortest path principle. The assumption for the setting is that if a real route exists between two IP nodes, these two IP nodes must be close or can be considered within the same IP network, ATM network should be avoided in this case. Link cost can also be used to favor or disregard specific links in order to achieve particular topology configurations.

4.2.2 User Specified Path vs. Shortest Path

TCP connections between source and destination agents are established by calling function *Simulator::connect(Agent* src, Agent* dst, Node** route=NULL, int cnt = 0).* The last two parameters are the user specified route path and the node count of this specified route. If they are not given, the system generated shortest route path will be used for the second layer routing. If they are past with the valid node list (user specified route path) and node count for all the nodes involved in the ATM switching, the user specified route path will be used for the second layer routing. Node lists are stored in the variable *Simulator::route_list_* which is indexed by TCP connection number. Node counts are stored in the fourth column of the global mapping table. A node list is valid means that all the nodes in the list should be either gateways or switches, all the neighboring nodes should have real links between them, node list should start and end

with gateways and has at least one switch in between. This feature is useful when a specific route path is required for ATM switching.

Since fully meshed gateway connection is used in HyNS, usually one gateway-gateway virtual link is needed when a switching path is generated by the system (shortest path). User specified path can contain more than one gateway-to-gateway virtual link for a TCP connection, as long as all the switches and gateways are specified when the connection is setup.

Function *Simulator::create_vc(int conn, int g1, int g2)* sets up the second layer routing (ATM switching) for TCP connection *conn* between gateway *g1* and gateway *g2*. The function is defined as follows:

```
void Simulator::create_vc(int conn, int g1, int g2)
{
        int nh;         // next hop
        int th;         // this hop
        int i;
        int found, index1, index2;
        int user_specified_path = 0;
        Link* link;

        RouteLogic* r = get_atm_routelogic();       // second layer route logic

        // get previously stored vci, user specified path and node count
        int vci = connection_[conn][CONN_VC];
        Node** route = route_list_[conn];
        int cnt = connection_[conn][CONN_ROUTE_CNT];
        if (route != NULL) {
                // search for g1 and assign its index to index1
                found = 0;
                i = 0;
                index1 = -1;
                while (i < cnt && found == 0) {
                        if (route[i]->id() == g1) {
                                found = 1;
                                index1 = i;
                        }
```

```
                i++;
        }
        if (found == 1) {
                // search for g2 and assign its index to index2
                found = 0;
                i = index1+1;
                index2 = -1;
                while (i < cnt && found == 0) {
                        if (route[i]->id() == g2) {
                                found = 1;
                                index2 = i;
                        }
                        i++;
                }
        }

        // checks the validity of the user specified path
        if (index1 != -1 && index2 != -1 && (index2-index1) != 1) {
                user_specified_path = 1;
                th = g1;                        // start from source gateway
                for(i=index1+1; i<=index2; i++) {
                        nh = route[i]->id();
                        link = link_[th][nh];
                        if (Node_[th]->type_ == IP_NODE ||
                            Node_[nh]->type_ == IP_NODE ||
                            link == NULL)
                                user_specified_path = 0;

                        th = nh;
                }
        }
}

// use user specified path if valid, otherwise use the shortest path
// generated by the system, populate switch table for each gateway or
// switch along the path
if (user_specified_path == 1) { // user specified path
        th = g1;                        // start from source gateway
        for(i=index1+1; i<=index2; i++) {
                nh = route[i]->id();
                link = link_[th][nh];
                if (link != NULL)
                        Node_[th]->add_switch(vci, link->head());

                th = nh;
        }
```

```
        }
        else { // shortest path
          th = g1;              // start from source gateway
          nh = r->lookup(th, g2);
          while (nh >= 0 && th != g2) {
            link = link_[th][nh];
                if (link != NULL)
                  Node_[th]->add_switch(vci, link->head());

                th = nh;
                nh = r->lookup(th, g2);
          }
        }
    }
```

Function *create_vc()* retrieves previously stored virtual circuit ID, user specified path and node count if specified, and checks the validity of the user specified path, and then populates switch table for each gateway or switch along the user specified path if the path is valid, otherwise along the shortest path generated by the system.

4.3 ATM Switch

ATM switches uses the Virtual Circuit Identifier (VCI) in each ATM cell header to determine how to route the cell. The typical structure of an ATM switch is as shown in Figure 4.

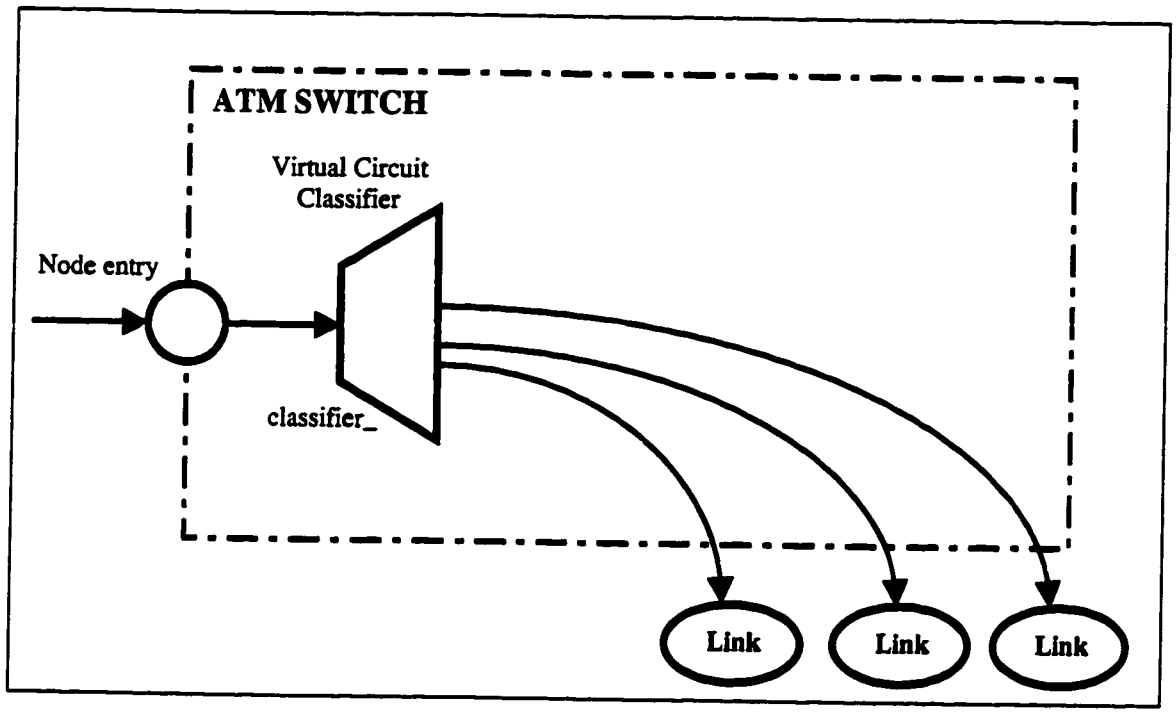

Figure 4 Typical Structure of an ATM Switch

4.3.1 Virtual Circuit Classifier

A virtual circuit classifier is used for cell forwarding, it is the first element that handles the incoming cells to the switch. It applies a bit-wise shift and mask operation to a cell's virtual circuit ID to produce a slot number. The slot number is returned from the *classify*() method. The class *VirtualCircuitClassifier* is defined as follows:

```
class VirtualCircuitClassifier : public Classifier {
public:
        VirtualCircuitClassifier() : mask_(~0), shift_(0) {
                mask_ = Classifier_mask_;
                shift_ = Classifier_shift_;
                off_cell_ = PacketHeaderManager::vartab_[OFF_CELL];
        }

        nsaddr_t mask_;
        int shift_;
```

```
protected:
        int off_cell_;
        int classify(Packet *const p) {
                hdr_cell* h = (hdr_cell*)p->access(off_cell_);
                return ((h->vci() >> shift_) & mask_);
        }
};
```

4.4 IP/ATM Gateway

A gateway is a computer that connects two different networks. An IP/ATM gateway connects IP and ATM networks, it segments an IP packet into ATM cells or reassemble ATM cells back into an IP packet. The typical structure of a gateway is shown in Figure 5.

In HyNS, each gateway contains a packet type classifier, which is the first element that handles the incoming packets to the gateway. Each packet type classifier contains an address classifier for routing packets and a virtual circuit classifier for switching cells. When a gateway receives a packet, its packet type classifier checks the incoming packet type. If the packet is an IP packet, the TCP connection to virtual circuit mapping table is checked. If a mapping is found, the gateway segments the IP packet into ATM cells and forwards them to its switch (or, virtual circuit classifier) which in turn forwards the cells to downstream ATM switch. This is the normal behavior of gateway. If the mapping is not found, the gateway forwards the packet to its router (or, address classifier) which in turn forwards the packet to the downstream IP node. Here, the gateway behaves like an IP router. If the packet is an ATM cell, the virtual circuit to TCP connection mapping table is checked.

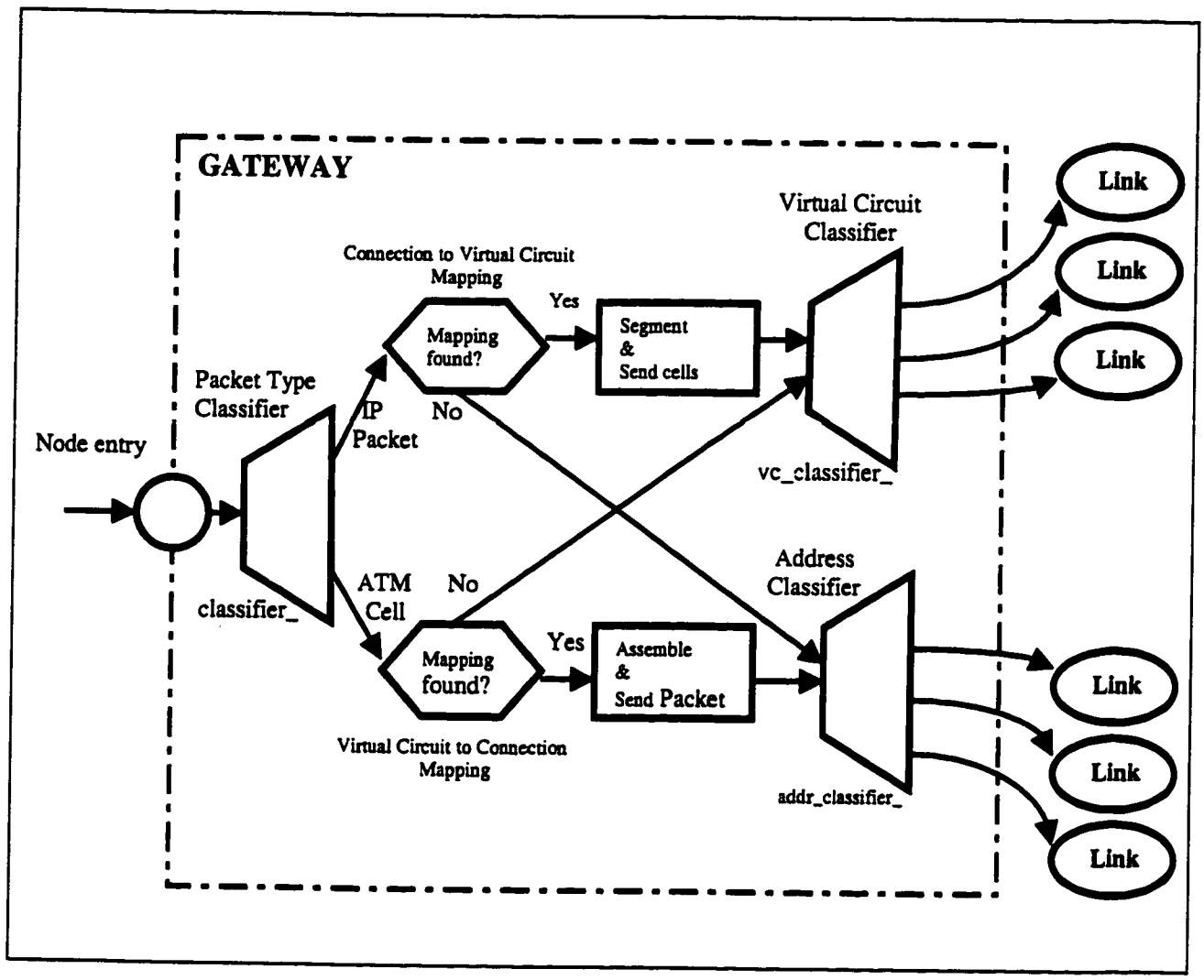

Figure 5 Typical Structure of a gateway

If the mapping is found and the cell is not the last one, the gateway buffers the cell. Or if the mapping is found, and the cell is the last one and all the cells are received in order, the gateway then reassembles all the cells received into IP packet and forwards the packet to its router (or address classifier) which in turn forwards the packet to the downstream IP node. This is the normal behavior of gateway. If mapping is not found, the gateway forwards the packet to its switch – virtual circuit classifier – which in turn forwards the cell to the downstream ATM switch, here gateway behaves like an ATM switch.

4.4.1 Packet Type Classifier

PacketTypeClassifier object is the first element that handles the incoming packets to the gateway, it is defined as follows:

```
class PacketTypeClassifier : public Classifier {
public:
        PacketTypeClassifier(int);
        void recv(Packet*, Handler* h = 0);

        inline AddressClassifier* address_classifier() {
                                        return (addr_classifier_);}
        inline VirtualCircuitClassifier* vc_classifier() {return (vc_classifier_);}
        int tcp2vc_[MAX_CONNECTIONS]; // TCP to VC mapping table
        int vc2tcp_[MAX_CONNECTIONS]; // VC to TCP mapping table
        // prints above two tables, route and switch tables
        void print_mapping_table();
        nsaddr_t mask_;
        int shift_;
        int off_cell_;

protected:
        AddressClassifier* addr_classifier_;        // address classifier
        VirtualCircuitClassifier* vc_classifier_;   // virtual circuit classifier
        int classify(Packet *const p);
        void ip2atm(Packet*);           // segmentation (IP packet to ATM cells)
        void atm2ip(Packet*);           // reassemble (ATM cells to IP packet)
        void add_cell(Packet*);
        int remove_cell(int vci, int last_seq);
        CellList* buffer_;              // keeps all the incoming cells

};
```

When a gateway receives a packet, the *recv()* function is called to check the incoming packet type. If incoming packet is an IP packet, the *ip2atm()* function is called; If incoming packet is an ATM cell, the *atm2ip()* function is called.

4.4.2 Segmentation

Member function *PacketTypeClassifier::ip2atm()* is called mainly to segment the IP packets into ATM cells and forward them to the downstream ATM switch, and it is defined as follows:

```
void PacketTypeClassifier::ip2atm(Packet* p)
{
        int i, conn, vci, ip_size, cell_num;
        hdr_cell* h;
        hdr_cell* h_cell;
        hdr_cmn* h_cmn;
        Packet* cell;

        // map TCP connection to ATM virtual circuit
        h_cell = (hdr_cell*)p->access(off_cell_);
        conn = h_cell->conn();
        vci = -1;
        if(conn != -1)
                vci = tcp2vc_[conn];

        if(vci == -1){
          // no mapping, send through gateway's router
          addr_classifier_->recv(p);
        }
        else { // segment and send through gateway's switch
          h_cmn = (hdr_cmn*)p->access(off_cmn_);
          h_cell->ip_ptype() = h_cmn->ptype();     // keep ip packet type
          h_cell->ip_size() = h_cmn->size();       // keep ip packet size
          ip_size = h_cmn->size();          // get simulated ip packet size
          h_cmn->ptype() = PT_CELL;
          h_cmn->size() = CELL_SIZE;
          h_cell->vci() = vci;
          h_cell->lastCellBit() = 0;

          cell_num = ip_size/CELL_DATA_SIZE + 1;

          // copy and send all but the last cell
          for(i=1; i<cell_num; i++) {
                cell = p->copy();
                h = (hdr_cell*)cell->access(off_cell_);
                h->seq() = i;
                vc_classifier_->recv(cell);
          }

          // send last cell
```

```
            h_cell->seq() = cell_num;
            h_cell->lastCellBit() = 1;
            vc_classifier_->recv(p);
        }
    }
```

This function retrieves the TCP connection number from the packet header, and looks up a mapping virtual circuit number from table *tcp2vc_*. If no mapping is found, the IP packet is forwarded through address classifier, here gateway behaves like an IP router; If a mapping virtual circuit is found, the IP packet is segmented into ATM cells and forwarded through virtual circuit classifier, here gateway behaves like a normal gateway. Each ATM cell has a sequence number and last cell bit field stored in the cell header. Sequence number is used to insure the cells are received in the original order before being reassembled back to the IP packet. The last cell bit field of the last cell is set to 1, and others set to 0.

4.4.3 Reassemble

Member function *PacketTypeClassifier::atm2ip()* is called mainly to buffer a cell or reassemble cells into IP packet and forward it to the downstream IP node, it is defined as follows:

```
void PacketTypeClassifier::atm2ip(Packet* cell)
{
        hdr_cmn* h_cmn = (hdr_cmn*)cell->access(off_cmn_);
        hdr_cell* h_cell = (hdr_cell*)cell->access(off_cell_);

        // map virtual circuit ID  to TCP connection number
        int conn = -1;
        int vci = h_cell->vci();
        if (vci != -1)
                conn = vc2tcp_[vci];

        if (conn == -1) {
                // no mapping, send through gateway's switch
```

```
                vc_classifier_->recv(cell);
        }
        else { // reassemble and send through gateway's router
                int last_cell_bit = h_cell->lastCellBit();
                if (last_cell_bit == 1) {               // last cell received
                        int last_seq = h_cell->seq();

                        // remove the previous buffered cells
                        int out_of_order = remove_cell(vci, last_seq);

                        // if all the cells received in order,
                        // reassemble and send the packet
                        if (out_of_order != 1) {
                                h_cmn->ptype() = h_cell->ip_ptype();
                                h_cmn->size() = h_cell->ip_size();
                                addr_classifier_->recv(cell);
                        }
                        else
                                Packet::free(cell);
                }
                else // buffer this cell
                        add_cell(cell);
        }
}
```

This function retrieves the virtual circuit number from the packet header, and looks up a mapping TCP connection number from table *vc2tcp_*. If mapping is not found, the cell is forwarded through virtual circuit classifier, here gateway behaves like a switch. If a mapping TCP connection is found, the cell is the last one from a packet and all the cells received in order, the cells will be reassemble back into the IP packet and forwarded through address classifier, here gateway behaves like a normal gateway. If the cells are received out of order, they will be discarded. If the cell is not the last one, simply buffer it.

5. A Simulation Example

A simple example is used to explain the usage of the hybrid network simulator. In this simple example, the agents in IP nodes from one IP network (node 0 and node 1) send constant bit rate packets to the agents in IP nodes in another IP network (node 3), passing through a high speed ATM network. The topology is shown in Figure 6.

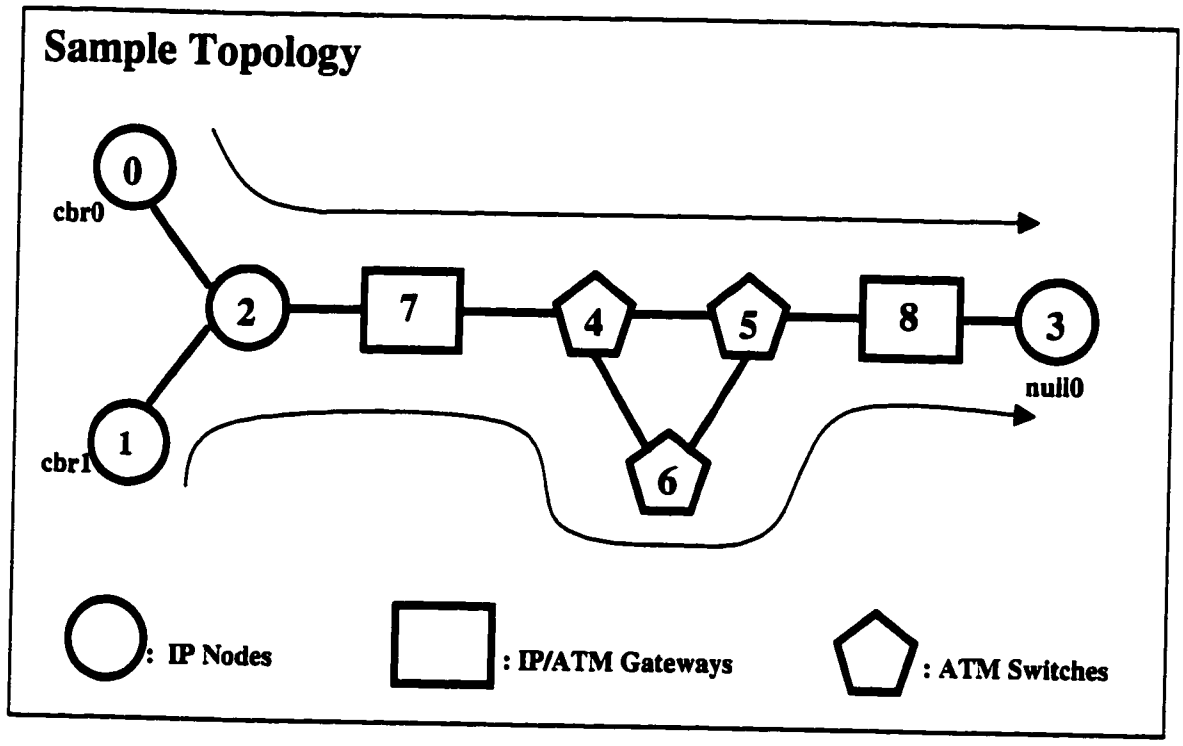

Figure 6 Sample Topology

5.1 Creating the Topology

The following code segment generates the topology shown in Figure 6:

```
Simulator* ns = new Simulator;

Node* n0 = ns->node(IP_NODE);
Node* n1 = ns->node(IP_NODE);
Node* n2 = ns->node(IP_NODE);
```

```
Node* n3 = ns->node(IP_NODE);
Node* n4 = ns->node(ATM_SWITCH);
Node* n5 = ns->node(ATM_SWITCH);
Node* n6 = ns->node(ATM_SWITCH);
Node* n7 = ns->node(GATEWAY);
Node* n8 = ns->node(GATEWAY);

ns->duplex_link(n0, n2, 5000000, 0.002, DROP_TAIL);
ns->duplex_link(n1, n2, 5000000, 0.002, DROP_TAIL);
ns->duplex_link(n2, n7, 1500000, 0.002, DROP_TAIL);
ns->duplex_link(n7, n4, 1500000, 0.01, DROP_TAIL);
ns->duplex_link(n4, n5, 1500000, 0.01, DROP_TAIL);
ns->duplex_link(n4, n6, 1500000, 0.01, DROP_TAIL);
ns->duplex_link(n5, n6, 1500000, 0.01, DROP_TAIL);
ns->duplex_link(n5, n8, 1500000, 0.01, DROP_TAIL);
ns->duplex_link(n8, n3, 1500000, 0.01, DROP_TAIL);
```

The above code segment first creates an instance of class *Simulator*, and calls function *node(int type)* to create four IP nodes (n0, n1, n2, n3), three ATM switches (n4, n5, n6) and two IP/ATM gateways (n7, n8) and then calls its member function *duplex_link(Node* n1, Node* n2, double bw, double delay, int queue_type)* to construct bi-directional links between node *n1* and node *n2* with bandwidth *bw* (bps) and delay *delay* (second), using queue of type queue_type.

5.2 Creating the Agents

The following code segment creates agents, attaches them to nodes, and establishes connection between source and destination agents:

```
CBR_Agent* cbr0 = new CBR_Agent;
ns->attach_agent(n0, cbr0);

CBR_Agent* cbr1 = new CBR_Agent;
ns->attach_agent(n1, cbr1);

NullAgent* null0 = new NullAgent(0);
ns->attach_agent(n3, null0);

Node* route[] = {n7, n4, n6, n5, n8};
```

```
int cnt = sizeof(route)/sizeof(route[0]);  // array length
ns->connect(cbr0, null0);
ns->connect(cbr1, null0, route, cnt);
```

The above code segment first creates two Constant Bit Rate (CBR) agents (crb0 and cbr1), and attaches cbr0 to node n0, cbr1 to node n1. Creates a null agent (null0) as traffic sink and attaches it to node n3, and then establishes two TCP connections between CBR agents and null agent. The connection between cbr0 and null0 uses the shortest path for second layer routing since last two parameters are not given. The connection between cbr1 and null0 uses user specified path for second layer routing, route sequence between gateway n7 and gateway n8 is n7-> n4-> n6-> n5-> n8.

5.3 Starting and Finishing the Simulation

The following code segment schedules three events, and starts the simulation by calling function *run()*:

```
ns->at(1.0, cbr_start, cbr0);
ns->at(1.1, cbr_start, cbr1);
ns->at(1.2, finish, ns);
ns->run();
```

Three scheduled events are:

- cbr0 starts sending packets at 1.0 (simulation time in seconds)
- cbr1 starts sending packets at 1.1 (simulation time in seconds)
- simulation ends at 1.2 (simulation time in seconds)

5.4 Trace Output

When above simulation is run, each individual packet is recorded as it arrives, departs, or is dropped at a link or queue. All these trace records are stored in a trace file to be post-processed and analyzed. Analyzing the trace file is one way of testing the simulator.

The trace output segments from the above simulation are listed and explained in this section. The first segment shows the travel path of a single packet from its source agent cbr0 to its destination agent null0, through the shortest path generated by the system. The second segment shows the travel path of a single packet from its source agent cbr1 to its destination agent null0, through the user-specified path (n7-> n4-> n6-> n5-> n8). Assuming the link cost for all the real (physical) links is one.

5.4.1 Trace Record Fields

In order to understand the trace output, the trace record fields are explained here. Each trace record (row) includes following fields separated by a space:

<code><time><hsrc><hdst><type><size><flags>
<flowID><src.sport><dst.dport><seq><pktID>
where
<code>: + (en-queue): packet entering queue;
- (de-queue): packet leaving queue
<time>: simulation time in seconds at which each event occurred
<hsrc>: first (source) node ID
<hdst>: second (destination) node ID
<type>: descriptive name for the packet type
<size>: packet's size, as encoded in its IP header
<flags>: six characters represent special flag bits which may be enabled.
The default value is "------".
<flowID>: IP flow identifier field as defined for IP version 6.1.
<src.sport>: packet source node ID & port number
<dst.dport>: packet destination node ID & port number
<seq>: packet sequence number (unique only to each source or to each packet when used in segmented cells)
<pktID>: unique packet identifier (for all the packets sent by all the sources)

An example of a trace file might appear as follows:

+ 1 0 2 cbr 210 ------ 0 0.0 3.0 0 0
- 1 0 2 cbr 210 ------ 0 0.0 3.0 0 0
+ 1.00234 2 7 cbr 210 ------ 0 0.0 3.0 0 0
- 1.00234 2 7 cbr 210 ------ 0 0.0 3.0 0 0
+ 1.00375 0 2 cbr 210 ------ 0 0.0 3.0 1 1

- 1.00375 0 2 cbr 210 ------ 0 0.0 3.0 1 1

Here we see six trace entries, 3 en-queue operations (indicated by "+" in the first column), 3 de-queue operations (indicated by "-"). The simulated time (in seconds) at which each event occurred is listed in the second column. The next two fields indicate between which two nodes tracing is happening. The next field is a descriptive name for the type of packet seen. The next field is the packet's size, as encoded in its IP header. The next six characters represent special flag bits that may be enabled. The next field gives the IP flow identifier field as defined for IP version 6.1. The subsequent two fields indicate the packet's source and destination node addresses, respectively. The following field indicates the sequence number. The last field is a unique packet identifier. Each new packet created in the simulation is assigned a new, unique identifier.

5.4.2 Shortest Path Packet Trace

The connection between cbr0 and null0 uses the shortest path generated by the system for second layer routing, the route sequence between two gateways should be n7->n4->n5->n8, the complete route should be n0->n2->n7->n4->n5->n8->n3. Now let's check the trace to see if packets really go through this route and been segmented and reassembled at the right node.

Following trace records the lifetime of the first packet sent from CBR agent cbr0 to null agent null0:

- At time 1 (in seconds), a CBR packet sent from node 0 to node 2 (en-queued and de-queued). CBR packet size is 210. Packet source node ID & port number is 0.0. Packet destination node ID & port number is 3.0. This packet is the first packet send by cbr0 (second to the last field is 0) and the first packet send by the simulator (last field is 0).

+ 1 0 2 cbr 210 ------ 0 0.0 3.0 0 0
- 1 0 2 cbr 210 ------ 0 0.0 3.0 0 0

- At time 1.00234, this CBR packet is forwarded from node 2 to node 7.

+ 1.00234 2 7 cbr 210 ------ 0 0.0 3.0 0 0
- 1.00234 2 7 cbr 210 ------ 0 0.0 3.0 0 0

- From time 1.00546 to time 1.00659, this CBR packet is segmented into 5 ATM cells in node 7 (IP/ATM gateway) and forwarded to node 4 (ATM switch). Notice that the packet type is changed from 'cbr' to 'cell', packet size is changed from 210 to 53 (cell size) and the packet sequence number 0 (second to the last field) is changed to cell IDs (1 to 5).

+ 1.00546 7 4 cell 53 ------ 0 0.0 3.0 1 0
- 1.00546 7 4 cell 53 ------ 0 0.0 3.0 1 0
+ 1.00546 7 4 cell 53 ------ 0 0.0 3.0 2 0
+ 1.00546 7 4 cell 53 ------ 0 0.0 3.0 3 0
+ 1.00546 7 4 cell 53 ------ 0 0.0 3.0 4 0
+ 1.00546 7 4 cell 53 ------ 0 0.0 3.0 5 0
- 1.00574 7 4 cell 53 ------ 0 0.0 3.0 2 0
- 1.00602 7 4 cell 53 ------ 0 0.0 3.0 3 0
- 1.0063 7 4 cell 53 ------ 0 0.0 3.0 4 0
- 1.00659 7 4 cell 53 ------ 0 0.0 3.0 5 0

- From time 1.015744 to time 1.01687, these 5 cells are forwarded from node 4 to node 5.

+ 1.01574 4 5 cell 53 ------ 0 0.0 3.0 1 0
- 1.01574 4 5 cell 53 ------ 0 0.0 3.0 1 0
+ 1.01602 4 5 cell 53 ------ 0 0.0 3.0 2 0
- 1.01602 4 5 cell 53 ------ 0 0.0 3.0 2 0
+ 1.0163 4 5 cell 53 ------ 0 0.0 3.0 3 0
- 1.0163 4 5 cell 53 ------ 0 0.0 3.0 3 0
+ 1.01659 4 5 cell 53 ------ 0 0.0 3.0 4 0
- 1.01659 4 5 cell 53 ------ 0 0.0 3.0 4 0
+ 1.01687 4 5 cell 53 ------ 0 0.0 3.0 5 0
- 1.01687 4 5 cell 53 ------ 0 0.0 3.0 5 0

- From time 1.02602 to time 1.02715, these 5 cells are forwarded from node 5 to node 8.

```
+ 1.02602 5 8 cell 53 ------ 0 0.0 3.0 1 0
- 1.02602 5 8 cell 53 ------ 0 0.0 3.0 1 0
+ 1.0263 5 8 cell 53 ------ 0 0.0 3.0 2 0
- 1.0263 5 8 cell 53 ------ 0 0.0 3.0 2 0
+ 1.02659 5 8 cell 53 ------ 0 0.0 3.0 3 0
- 1.02659 5 8 cell 53 ------ 0 0.0 3.0 3 0
+ 1.02687 5 8 cell 53 ------ 0 0.0 3.0 4 0
- 1.02687 5 8 cell 53 ------ 0 0.0 3.0 4 0
+ 1.02715 5 8 cell 53 ------ 0 0.0 3.0 5 0
- 1.02715 5 8 cell 53 ------ 0 0.0 3.0 5 0
```

- At time 1.03743, these 5 cells are reassembled back to CBR packet in node 8 (IP/ATM gateway) and forwarded to node 3, the destination node. Notice that the packet type is changed from 'cell' back to 'cbr', packet size is changed from 53 back to 210 and the cell IDs are changed back to packet sequence number 0.

```
+ 1.03743 8 3 cbr 210 ------ 0 0.0 3.0 0 0
- 1.03743 8 3 cbr 210 ------ 0 0.0 3.0 0 0
```

We can see from the above trace analysis that the first packet sent from cbr0 actually travels through the shortest path generated by the system and been segmented and reassembled at the right node.

5.4.3 User Specified Path Packet Trace

The connection between cbr1 and null0 uses user specified path for second layer routing, the route sequence between two gateways is n7->n4->n6->n5->n8, the complete route should be n1->n2->n7->n4->n6->n5->n8->n3. Now let's check the trace to see if packets really go through this route and been segmented and reassembled at the right node.

Following trace records the lifetime of the first packet sent from CBR agent cbr1 to null agent null0:

- At time 1.1 (in seconds), a CBR packet sent from node 1 to node 2. Packet source node ID & port number is 1.0. Packet destination node ID & port number is 3.0. This packet is the first packet send by cbr1 (second to the last field is 0) and the 27th packet send by the simulator (last field is 27).

```
+ 1.1 1 2 cbr 210 ------ 0 1.0 3.0 0 27
- 1.1 1 2 cbr 210 ------ 0 1.0 3.0 0 27
```

- At time 1.10234, this CBR packet is forwarded from node 2 to node 7.

```
+ 1.10234 2 7 cbr 210 ------ 0 1.0 3.0 0 27
- 1.10234 2 7 cbr 210 ------ 0 1.0 3.0 0 27
```

- From time 1.10546 to time 1.10659, this CBR packet is segmented into 5 ATM cells in node 7 (IP/ATM gateway) and forwarded to node 4 (ATM switch). Notice that the packet type is changed from 'cbr' to 'cell', packet size is changed from 210 to 53 (cell size) and the packet sequence number 0 (second to the last field) is changed to cell IDs (1 to 5).

```
+ 1.10546 7 4 cell 53 ------ 0 1.0 3.0 1 27
- 1.10546 7 4 cell 53 ------ 0 1.0 3.0 1 27
+ 1.10546 7 4 cell 53 ------ 0 1.0 3.0 2 27
+ 1.10546 7 4 cell 53 ------ 0 1.0 3.0 3 27
+ 1.10546 7 4 cell 53 ------ 0 1.0 3.0 4 27
+ 1.10546 7 4 cell 53 ------ 0 1.0 3.0 5 27
- 1.10574 7 4 cell 53 ------ 0 1.0 3.0 2 27
- 1.10602 7 4 cell 53 ------ 0 1.0 3.0 3 27
- 1.1063 7 4 cell 53 ------ 0 1.0 3.0 4 27
- 1.10659 7 4 cell 53 ------ 0 1.0 3.0 5 27
```

- From time 1.115744 to time 1.11687, these 5 cells are forwarded from node 4 to node 6.

+ 1.11574 4 6 cell 53 ------ 0 1.0 3.0 1 27
- 1.11574 4 6 cell 53 ------ 0 1.0 3.0 1 27
+ 1.11602 4 6 cell 53 ------ 0 1.0 3.0 2 27
- 1.11602 4 6 cell 53 ------ 0 1.0 3.0 2 27
+ 1.1163 4 6 cell 53 ------ 0 1.0 3.0 3 27
- 1.1163 4 6 cell 53 ------ 0 1.0 3.0 3 27
+ 1.11659 4 6 cell 53 ------ 0 1.0 3.0 4 27
- 1.11659 4 6 cell 53 ------ 0 1.0 3.0 4 27
+ 1.11687 4 6 cell 53 ------ 0 1.0 3.0 5 27
- 1.11687 4 6 cell 53 ------ 0 1.0 3.0 5 27

- From time 1.12602 to time 1.12715, these 5 cells are forwarded from node 6 to node 5.

+ 1.12602 6 5 cell 53 ------ 0 1.0 3.0 1 27
- 1.12602 6 5 cell 53 ------ 0 1.0 3.0 1 27
+ 1.1263 6 5 cell 53 ------ 0 1.0 3.0 2 27
- 1.1263 6 5 cell 53 ------ 0 1.0 3.0 2 27
+ 1.12659 6 5 cell 53 ------ 0 1.0 3.0 3 27
- 1.12659 6 5 cell 53 ------ 0 1.0 3.0 3 27
+ 1.12687 6 5 cell 53 ------ 0 1.0 3.0 4 27
- 1.12687 6 5 cell 53 ------ 0 1.0 3.0 4 27
+ 1.12715 6 5 cell 53 ------ 0 1.0 3.0 5 27
- 1.12715 6 5 cell 53 ------ 0 1.0 3.0 5 27

- From time 1.1363 to time 1.13748, these 5 cells are forwarded from node 5 to node 8.

+ 1.1363 5 8 cell 53 ------ 0 1.0 3.0 1 27
- 1.13635 5 8 cell 53 ------ 0 1.0 3.0 1 27
+ 1.13659 5 8 cell 53 ------ 0 1.0 3.0 2 27
- 1.13663 5 8 cell 53 ------ 0 1.0 3.0 2 27
+ 1.13687 5 8 cell 53 ------ 0 1.0 3.0 3 27
- 1.13691 5 8 cell 53 ------ 0 1.0 3.0 3 27
+ 1.13715 5 8 cell 53 ------ 0 1.0 3.0 4 27
- 1.1372 5 8 cell 53 ------ 0 1.0 3.0 4 27
+ 1.13743 5 8 cell 53 ------ 0 1.0 3.0 5 27
- 1.13748 5 8 cell 53 ------ 0 1.0 3.0 5 27

- At time 1.14776, these 5 cells are reassembled back to CBR packet in node 8 (IP/ATM gateway) and forwarded to node 3, the destination node. Notice that the packet type is changed from 'cell' back to 'cbr', packet size is changed from 53 back to 210 and the cell IDs are changed back to packet sequence number 0.

+ 1.14776 8 3 cbr 210 ------ 0 1.0 3.0 0 27
- 1.14776 8 3 cbr 210 ------ 0 1.0 3.0 0 27

We can see from the above trace analysis that the first packet sent from cbr1 actually travels through the user specified path and been segmented and reassembled at the right node.

6. Summary

Because of the lack of IP+ATM simulators, my goal was to develop a dual or hybrid network simulator. In order not to reinvent wheels, Lawrence Berkeley National Laboratory (LBNL)'s Network Simulator version 2 (Ns2) was chosen as the base simulator for building my IP/ATM Hybrid Network Simulator (HyNS).

Ns2 is written in C++ and OTcl, any one wish to fully understand or modify it must be comfortable with both environments and tools. Plus, it is hard to debug both C++ and OTcl code together. Since not many researchers are familiar with OTcl, we decided to take out the OTcl part and convert OTcl code into C++ code, this way, users of HyNS will only need to know C++ to understand the simulator, and use any standard C++-Level debugger for debugging.

Ns2 has good support for IP, but no support for ATM. In order to support ATM and TCP/IP over ATM, following components are added to the modified and translated Ns2:

- TCP connection to ATM virtual circuit mapping
- Second layer routing (ATM switching)
- ATM switch (Virtual Circuit Classifier)
- IP/ATM gateway function (segmentation and reassemble)

HyNS was tested before and after the addition of the above ATM components. In this hybrid simulator, most of the objects have a name field, and a lot of classes provide debugging functions, which print out the object information. The simulator also supports trace object which records each individual packet as it arrives, departs, or is dropped at a

link or queue. The simulator was tested with sample topologies using the above debugging supports. During the test process, some tables (IP and ATM routing tables, global and local TCP connection to ATM virtual circuit mapping tables) and scheduler were printed and checked manually against the sample topologies. When a packet is traced from its source node to its destination node, a list of names of the objects that received and forwarded the packet along the path was obtained. The trace file was also analyzed to make sure the simulator implementation is correct. Users of the simulator can also use these debugging supports to understand the simulator.

HyNS was developed to provide a means for researchers to analyze the behavior of IP networks or IP over ATM networks without the expense of building a real network. It can be used for testing various IP-over-ATM algorithms, investigating the performance of TCP connections over ATM networks without ATM-level congestion control, and comparing it to the performance of TCP over packet-based networks, etc.

Appendix A Conversion to Windows NT and Windows 95

HyNS was originally developed under Unix Solaris 2.5, and ported to Windows NT 4.0 and Windows 95. Nowadays, personal computers can be found every where, in the office or at home. So it is very convenient for the researchers to have a version of HyNS running under Windows environment like Windows NT and Windows 95.

The following are the steps to create HyNS console program using Microsoft Visual C++ 5.0 under Windows NT or Windows 95 environment:

1. Create a new project of type "Win32 Console Application"
   - In Microsoft Developer Studio, On the File menu, click New and then click the Projects tab.
   - Select "Win32 Console Application" as project type.
   - Specify the Project Name (HyNS), Location, and then click the OK button.
2. Rename all the *.cc files to *.cpp files
3. Add all the *.cpp files and *.h files to the new project space
   - On the Project menu, click "Add to project", and then click Files
   - Select all the *.cpp and *.h files, click OK.
4. Build and run the application
   - On the Build menu, click "Build HyNS.exe" to build the application.
   - On the Build menu, click "Execute HyNS.exe" to run the application.

Console programs are developed with Console API functions, which provide character-mode support in console windows. The Visual C++ run-time libraries also provide output and input from console windows with standard I/O functions, such as *printf()* and *scanf()*.

References

[FALL97] Kevin Fall, Kannan Varadhan, ns Notes and Documantation, November 6, 1997

http://www-mash.CS.Berkeley.EDU/ns/ns-documentation.html

[KUMA97] Satish Kumar, VINT Project Overview

http://netweb.usc.edu/vint/project_overview.html

[KESH95] S. Keshav, Carsten Lund, Steven Phillips, Nick Reingold, Huzur Saran, "An Empirical Evaluation of Virtual Circuit Holding Time Policies in IP-over-ATM Networks", IEEE Journal on Selected Areas in Communication, October 1995

[MAH96] Bruce A. Mah, An Internet Simulated ATM Networking Environment (INSANE) Users Manual, The Tenet Group Computer Science Division University of California at Berkeley

http://www.cs.berkeley.edu/~bmah/Software/Insane/

[MCCA97] Steven McCanne, Sally Floyd, Network Simulator Version 2 Manual, University of California, Berkeley and Lawrence Berkeley National Laboratory, Berkeley.

http://www-mash.cs.berkeley.edu/ns/ns-man.html

[OTcl95] OTcl Tutorial (Version 0.96, September 95)

ftp://ftp.tns.lcs.mit.edu/pub/otcl/doc/tutorial.html

[ROMA94] A. Romanow and S. Floyd, The Dynamics of TCP Traffic over ATM Networks, in SIGCOMM Symposium on Communications Architectures and Protocols, (London, UK), pp. 79-88, Sept. 1994. Also available by anonymous ftp from playground.sun.com: pub/tcp_atm/tcpatm_extended.*.ps.

[SHAH97] Rawn Shah, Understanding ATM networking and network layer switching, part one. Learn about the issues behind ATM and TCP/IP integration: How is it done? How will vendors do it in the future?

http://sunsite.icm.edu.pl/sunworldonline/connectivity.html

[STAL97] William Stallings, High-Speed Networks: TCP/IP and ATM Design Principles.

[SATY98] Ramakrishna V. Satyavolu, IP over ATM, Term project report for the course Broadband Networks offered by Professor Kenneth Vastola. Feb. 6, 1998