Programmers Manual For TCP/IP Over ATM Simulator

Introduction

This document is intended for a those who want to change this simulator or add any new component to this simulator. It is essential to understand how different functionalities of the simulator are achieved by various classes. Not only will this ensure a valid addition to the design, but can also ease the programming effort by using the predefined classes. We believe that the source code is the best guide to understanding a program but for a simulator of this nature, knowing the basics of the design greatly enhance the understanding of the program. So we have included both the description of the classes and parts of their code. If you have not read the "Basic Design Document" then read it first. The following is a detailed description of the classes. For every class we first brifly describe the class, then its individual members and important functions.

TclObject (Tcl.h, Tcl.cc)

TclObject is the highest class in the object hierarchy. All the major classes are derived from this class. The advantage of using this common base class is that objects of all kinds can be stored in the same list .

Variables

id_	A static integer that assigns unique names to all the TclObjects
all_	A static list of all the TclObjects. Every object that is inherited from TclObject will be inserted in this list.
name_	This is the unique name of the object. For user defined objects the name is **"_on"** where **n** is the unique number sequentially assigned by id_. For other objects i.e. command objects, the names can be specified by the programmer. e.g. "**ns-at"**
class_name_	This contains the class names as identified by the Matcher classes. If it belongs to a hierarchy of classes then it will contain both of them seperated by slash. For e.g. **"link/atm"** for atmlinks.
next_	This is for maintaining the linked list of the objects.

Important Functions

lookup	It takes a name and returns the pointer to the TclObject with that name. The pointer can be typecast to satisfy the needs of the function. This command is particulalrly useful for identifying an object from the user input.
reset_all	Resets all the objects. Reset is a virtual function and every object can make its own reset function. This is useful for resetting the statistics so that statistics can be gathered starting at any time.

Another **pure virtual** function is **command** which should be defined for any inheriting class and for handling its input the way it wants to.

Special Commands (Tcl.cc, misc.cc)

These commands directly inherit from the TclObject class. They are all static. Thus only one instance of this object can be created. Not only this the object is created along with it. However they are assigned special names. For example the **NowCommand** is identified by the "**ns-now**" string as shown.

```
static class NowCommand : public TclObject {
public:
        NowCommand() : TclObject("ns-now") { }
        virtual int command(int argc, const char*const* argv);
} now_cmd;
```

These commands include :

DeleteCommand

The **DeleteCommand** can be used to delete a TclOject. This not only frees the memory but also removes it from the linked list of all the objects .This is achieved through the TclObject Destructor. The destructor itself is a virtual function and can be over-ridden by its inherited classes. **DeleteCommand** itself a TclObject with the name **"delete"** .

CreateCommand

The **CreateCommand** can be used to create an object of any class and id. **CreateCommand** is a class inherited from TclObject with the name **"new".** It looks up the class from the Matcher classes and a new object of that class is created. All objects are created using the **new** command.

AtCommand

The **AtCommand** is used to schedule events at a certain time from the user input. It takes in the time and a valid Tcl command and generates an **AtEvent** at that time. Any valid command can be supplied as a string. For example

```
ns at 0.0 "$ftp start"
```

This command is interpreted by the **ns procedure** in the ns-default.tcl file, which invokes the AtCommand function.

NowCommand

The **NowCommand** is identified by the name **"ns-now"** . It is interpreted by the **ns procedure.** This invokes the NowCommand which in turn returns the current time. The time is returned as a string in the Tcl buffer.

RunCommand

The **RunCommand** is used to actually run the simulation. It is identified by the **"ns-run"** command. When the user input is **ns run** it is interpreted by the ns-default file through the **ns procedure.** As a result of which the RunCommand function gets executed. This function resets all the statistics (possibly from a previous simulation run) and then calls the run function of the scheduler class.

```
int RunCommand::command(int argc, const char*const* argv)
{
        Scheduler& s = Scheduler::instance();
        s.reset();
        TclObject::reset_all();
        s.run();
        return (TCL_OK);

}
```

RandomCommand

This command is used to generate Random numbers. it is identified with "**ns-random"** command. It can be used both with a seed and without.

Matcher

The Matcher classes are used to identify the class whose object has to be created. The Matcher class is the parent class of all the Matcher classes. For every class whose object need to be created, you can create the corresponding Matcher class with a chosen id for taking user input. Every Matcher class is declared static so that only one Matcher object can be created. Just like the TclObject, Matcher also stores all the objects in a static list of all the Matcher objects. ***For every new class whose object has to be created, a corresponding Matcher class must also be created if user input needs to be applied on it.***

Variable

all_	The static list of all the Matcher object
next_	Pointer to the next Matcher object.
classname_	The name of the class identifying the Matcher class

Functions

lookup	This function takes in the classname and the id and returns the TclObject of that class. The class is identified by the match function which returrns a new object of that type if the id matches. Lookup is called by the **CreateCommand.**
match	It is a virtual function which matches the classname and the id and returns a new object of that type if successful. It is called by lookup.

For example, the class **DropTailMatcher** is used to identify the **DropTailLink** class by comparing the id "**drop-tail".** The user can specify the type of the link as "**drop-tail"** as the input and using the Matcher class can create an object of type DropTailLink. An example :

```
static class ATMSwitchMatcher : public Matcher {
public:
        ATMSwitchMatcher() : Matcher("atm-switch") {}
        TclObject* match(const char* id) {
                if(strcasecmp(id, "output") == 0)
                        return (new ATMOutputBuffer());
                if(strcasecmp(id, "shared") == 0)
                        return (new ATMSharedBuffer());
                return (0);
        }
} matcher_atm_switch;
```

Class Handler

The Handler object is used as the base class for all the classes that need to handle events. All it contains is a pure virtual function which enforces that all future inherited classes must provide this function. So if one wants to introduce a new class which needs to handle events, then it must inherit from the Handler class. This is essential since an **Event** has a field **handler_** which points to the object that will handle the event.

```
     class Handler {
      public:
             virtual void handle(Event* event) = 0;
```

```
};
```

Class Event

The Event class represents the actual events that are generated by the simulator. It contains the time of the event and a pointer to the Handler object which will handle the event. By using this as the base class, Packet and AtHandler were defined. The advantage of this scheme is that whoever has to handle the event, converts the event to whatever form it wants to convert it to and then handles it accordingly.

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

Class AtEvent

The **AtEvent** class is used for handling Tcl commands at particular events. The command is stored as a string and is used by calling the Tcl eval function which interprets the command.

Class Scheduler

The **Scheduler** class is the main driver for the simulator. All events are scheduled using the **Scheduler** class. Its variables include :

clock_	The current time in the network. It is protected and so can not be tempered with.
queue_	A sorted list of all the events. They are sorted according to increasing time.
instance_	A static pointer to the scheduler. It will not be initialised during a simulation
uid_	A unique id for an event. Mostly useful for debugging purposes.

Member Functions

instance	Returns the instance of the scheduler. ***Scheduler::instance()*** is used to gain access to the clock or schedule an event etc.
clock	This function returns the current time.
cancel	This function looks up the uid of the event and removes it from the queue. The caller must free the event if necessary, this routine only removes it from the scheduler queue.

Scheduler::schedule()

The Schedule function takes the reference to an event, a handler and a delay in time units into the future when the event is to be generated. The user id for the event is generated and the event is dispatched to the specified handler. The event is assigned a time at which it is supposed to take place. The event is then assigned to the priority queue of events in which they are in ascending order of their times of generation. If any object of any class needs to generate an event it has to call the scheduler. It should provide with the handler object that will handle the event, the event (Packet, ATMCell etc.) and the delay from the current time.

```
void Scheduler::schedule(Handler* h, Event* e, double delay)
{
        e->uid_ = uid_++;
        e->handler_ = h;
        double t = clock_ + delay;
        e->time_ = t;
        /* XXX replace this code with Sugih's calendar queue */
        Event** p;
        for (p = &queue_; *p != 0; p = &(*p)->next_)
                if (t < (*p)->time_)
                        break;
        e->next_ = *p;
        *p = e;
}
```

Scheduler::run()

This function actually drives the simulation. It is called by the **RunCommand.** It takes an event from the event queue and handles it using the handler. The time is updated to be the time of the current event. If there are no more events to handle then it stops.

```
void Scheduler::run()
{
        while (queue_ != 0) {
                Event* p = queue_;
                queue_ = p->next_;
                clock_ = p->time_;
                p->handler_->handle(p);
        }
}
```

Packet (packet.h)

This class inherits from the **Event** class. This is so because a packet's movements in the network can be easily tracked by typecasting it to an event. Thus any object that has to handle packets can simply use the handle function. The packet includes the following fields:

type_	The type of the packet. The types are defined in packet.h. They include, **PT_ACK, PT_START, PT_STOP, PT_PRUNE, PT_GRAFT, PT_MEASSAGE, PT_NTYPE**
class_	This is used for router statistics. It is also used by Class Based Queueing
qtime_	It gives the time the packet was put on queue for the current hop
lsrc_	Link-level source. This is the address of the source of the current hop.
ldst_	Link-level destination. This is the address of the destination of the current hop.
src_	The source of the packet. Used by routers
dst_	The destination of the packet. Used by the routers for routing
sport_	The port number of the source agent.
dport_	The port number of the destination agent.
size_	The size of the packet in bytes
seqno_	The Sequence number of the packet
uidcnt_	A static variable for assigning unique ids for the packets.
uid_	A unique id for this packet

Packet::alloc() - Create a new packet.

Var (object.h, object.cc)

This class contains the objects of the default variables declared in the input file. It contains things like the name of the variable its characteristics etc. Whenever an object is created its default values can be set in the constructor by searching for the appropriate variables and class name.

name_	The name of the variable
readonly_	Whether the variable is readonly or not. Readonly variables in ns_default cannot be altered while other default variables can be altered during the simulation.

All the network default variables belong to one of these classes. There are basically two types of classes inheriting from the base class Var. VarReal and VarInt. You can add your own variables in the ns_default.tcl file but they will be characterised by one of these.

VarReal

This class inherits directly from the **Var** class. The VarReal class handles the real numbered values i.e. double. This class takes in the values assigned to real numbered variables and creates the VarReal object for them. It also takes in the names of these variables and assigns the types whether they are read only or not.

VarBandwidth

This class inherits from the VarReal class. This is because it takes in real numbers for the bandwidth of links etc.

```
link_bw("ns_link", "bandwidth", &bandwidth_, 0);
```

The main difference is that they convert values with units to double values. It uses a function ***bw_atof*** which takes in a string and converts it into a double. For e.g.

```
set ns_link(bandwidth) 1.5Mb
```

the 1.5Mb string is converted to 1,500,000. It understands k and K for 1000, m and M for 1000000 and B for bytes while b for bits.

VarTime

This class too inherits from the VarReal class. Again it can take in units of time and convert them to values. This is done by calling a function ***time_atof.*** Examples of such variables are

```
set ns_delsink(interval) 100ms
set ns_cbr(interval) 3.75ms
set ns_rlm(interval) 3.75ms
```

VarInt

This class also inherits from the Var class. It takes care of the whole numbers (integer values) that are assigned to default variables. Some examples are

```
set ns_sink(packet-size) 40
set ns_rlm(packet-size) 210
set ns_link(queue-limit) 50
```

VarBool

This class inherits from the VarInt class. As must be evident from the name it takes care of the default boolean values that are assigned to different variable e.g true or false. It simply looks at the first letter of the string and if it is T or t then it is assigned a value of 1 and 0 otherwise. Some examples are

```
set ns_class(plot) false
set ns_red(setbit) false
set ns_red(drop-tail) false
set ns_red(wait) true
```

NsObject (object.h, object.cc)

NsObject is the base class for all the entities in a Network. These include **Nodes, Links, Agents, Trace** and **DataSources**. **Node, Link** and **Agents** are derived from both the **Handler** and **NsObject.** The reason for this is that these 3 objects have to handle various events while the others do not have to. NsObject contains a list of Var objects which have been stored from ns_default.tcl file. So when an object inherited from the NsObject class is created the default values of its variables can be set using the appropriate function from these functions; link_real, link_bw, link_time, link_int and link_bool. This can be done in the constructor of the class. As all the Var objects are visible to all the objects, the user input can be easily translated into their respective values. Also one avoids the use fo hardcoded constructor values and can change the default values without recompiling the program. Another attribute of this class is the **Trace** object which if attached to an object can be used to trace its statistics. Other objects like various data sources and flow managers are also derived from NsObject. It contains the NsObject class which inherits from the TclObject class. The Var pointer of the var class is also defined.

This is responsible for handling the inputs from the ns_default.tcl file.

Functions On the NsObject Class

NsObject::NsObject()

The NsObject constructor sets the default values for the var and trace variables equal to 0.

void NsObject::insert(Var* p)

This object inserts the variable that is declared to the list of variables of the particular NsObject hence created. It takes in the pointer to the variable as an argument and returns nothing.

int NsObject::command(int argc, const char*const* argv)

This command function interprets the command in ns_default.tcl file. The NsObject can interpret the commands of **set** and **get.** i.e. setting and retrieving the values of the variable. If it is unable to handle the command it hands it over to its parent class **TclObject.** This is true for all the **command** functions in any class. If it can not interpret the command then try the higher class in the heirarchy. For example if a command does not match any of the cases in the link::command function so it is passed to the upper class in the object hierarchy i.e. the NsObject command function. Here it can match with the set command and the get command. The set case sets the values for the particular variable. The get case returns the value. If that too fails then hand over to the TclObject function. The trace case switches on the trace mode set for connections. The following commands of the link are matched in NsObject command function.

```
$link0 set bandwidth $bw
$link0 set delay $delay
int Link::command(int argc, const char*const* argv)
int NsObject::command(int argc, const char*const* argv)
```

Var* NsObject::varlookup(const char*var) const

As must be evident from the very name of this function. Given the name of the variable, it searches the name in the list and returns a pointer to the variable. In case it is not found it returns error that the particular object is not found.

Int NsObject::get(const char*var)

This function is invoked from the Command function (described above) after looking at the "set" keyword in the ns_default.tcl file. It invokes the varlookup function (described above as well) which matches the name whether it is there or not and returns the var object. The get function then eventually returns the tcl instance of this variable and its value. If the varlookup returns a null i.e. it did not find the name in the list it returns an error.

Int NsObject::set(const char*var, const char*val)

This function too is invoked by the Command function described earlier after matching with the "get" keyword in the ns_default.tcl file. It takes in two arguments, the name of the function and the value that is supposed to be set. It invokes the varlookup routine first to see whether the variable name exists in the list of variables or not. If yes then it checks to see whether its value is readonly or not, if it is readonly it does not set the value and prompts that it is readonly and hence cannot be modified. On the other hand if the value is not readonly it sets the value attribute of the variable.

void NsObject::link_real(const char* a, const char* name, double* val, int readOnly)

This function gets invoked from the different link files. For example in the file red.cc. It invokes the VarReal function which declares a variable that can take in real numbers. It also sets the values of the name and characteristics of that particular Tcl Instance belonging to the variable. It is usually called from the constructor to initialise the values to the default values.

```
link_real ("ns_red", "thresh", &edp_.th_min, 0);
link_real ("ns_red", "maxthresh", &edp_.th_max, 0);
```

void NsObject::link_bw(const char* a, const char* name, double* val, int readOnly)

This function gets invoked from the link files. For example the link.cc file. It inovkes the VarBandwidth function in return (which inherits from the VarReal function). This function returns the variable of type VarBandwidth. Then the set function is invoked which sets the name and value of that particular variable to the TclObject. Eventually the insert function is called which adds it to the list of variables.

```
link_bw("ns_link", "bandwidth", &bandwidth_, 0);
```

void NsObject::link_time(const char* a, const char* name, double* val, int readOnly)

This function gets invoked as a result of the following commands in the link.cc files. This function declares an object of type VarTime which inherits from the Class VarReal. This variable in turn is given to the set function which sets the name of the function and its value i.e. ns_link and delay values. It then invokes the insert function which inserts this variable in the list of variables.

```
link_time("ns_link", "delay", &delay_, 0)
```

void NsObject::link_int(const char* a, const char* name, double* val, int readOnly)

This function gets invoked as a result of the following commands in link.cc. This function declares an object of type VarInt which inherits from the class Var. This variable is turn is given to the set function which sets the name of the function and its values. It then invokes the insert function which inserts this variables in the list of variables.

```
link_int("ns_link", "queue-limit", &qlim_, 0);
link_int(0, "queue-length", &qnp_, 1);
link_int(0, "queue-size", &qnb_, 1);
```

void NsObject::link_bool(const char* a, const char* name, double* val, int readOnly)

This function gets invoked as a result of the following commands in red.cc. This function declares an object of type VarBool which inherits from the class VarInt. This variable is turn is given to the set function which sets the name of the function and its values. It then invokes the insert function which inserts this variables in the list of variables.

```
link_bool("ns_red", "bytes", &edp_.bytes, 0);
```

double bw_atof(const char*s)

This function converts the input at the ns_default.tcl file e.g

```
set ns_link(bandwidth) 1.5Mb
```

It takes it in and converts it into its reall units i.e., 1.5 million bits.

double time_atof(const char* s)

It takes the input time in the units and conversts it into the appropriate number. There are various case options available for it. m millisecod, u microsecond, n nanoseconds, p pecosecond. For e.g.

```
set ns_link(delay) 100ms
```

For an all round use of the NsObject class, here is the constructor for the REDQueue

```
REDQueue::REDQueue()
{
Tcl& tcl = Tcl::instance();

link_bool("ns_red", "bytes", &edp_.bytes, 0);
link_real("ns_red", "thresh", &edp_.th_min, 0);
link_real("ns_red", "maxthresh", &edp_.th_max, 0);
link_int("ns_red", "mean_pktsize", &edp_.mean_pktsize, 0);
link_real("ns_red", "q_weight", &edp_.q_w, 0);
link_bool("ns_red", "wait", &edp_.wait, 0);
link_real("ns_red", "linterm", &edp_.max_p_inv, 0);
link_bool("ns_red", "setbit", &edp_.setbit, 0);
link_bool("ns_red", "drop-tail", &drop_tail_, 0);

link_bool("ns_red", "doubleq", &doubleq_, 0);
link_int("ns_red", "dqthresh", &dqthresh_, 0);
}
```

Link (link.h, link.cc)

This class is the representation of the output link. A link is unidirectional. A node can have multiple output lines i.e. a router or a single output line (the source). However two nodes can not have more than one link. The link class inherits from the Handler and NsObject classes. The main attributes of the link class are

src_	The Starting end of the link
dst_	The Destination node of the link
qnp_	The Number of Packets in queue
qnb_	Number of bytes in the queue
qlim_	Maximum allowed packets in queue
busy_	It is true while a packet is being transmitted
neighbor_	The destination node. Used for scheduling the next event
bandwidth_	The bandwidth of the link in terms of bits
delay_	The latency in the link delay is given by delay it is in seconds
busytime_	Total time the link was busy
ib_	The integrator for the bytes
ip_	The integrator for packets
iu_	The integrator for utilization
qtrace_	For tracing the queue
flowmgr_	The flow manager if attached

There is a separate Queue class, which is incorporated into the link. The link is implemented as a queue since the out going buffers at any node are in directly unique for each link and hence can be considered a part of the link. The enqueue, dequeue, and qlen functions are declared as being virtual implying that all the classes inheriting from the link class will have to make their own functions regarding the above mentioned operations.

Functions

Link::Link()

The constructor for the link, looks up for the variables by using the **link_real, link_bw, link_int, link_time** functions and initializes the parameters such as the bandwidth, delay, queue limit, for the link with the values from the default file.

void Link::update(Packet * pkt, int how)

This function is called every time the queue size changes and updates the vital statistics of the queue.

void Link::send()

This function dequeues the packet from the link queue, calculates the transmission time required, adds the delay to it that has to go through on the link, updates the utilization. and eventually an event is generated for that packet to get off the link. It also sets the busy attribute to 1. Also if a trace object is attached to the function then it will record the hop for the packet. This function is called by the handle function

```
void Link::send()
{
        Packet* pkt = deque();
        if (pkt != 0) {
                double txtime = pkt->size_ * 8. / bandwidth_;
```

```
                Scheduler& s = Scheduler::instance();
                pkt->lsrc_ = src_;
                pkt->ldst_ = dst_;
                s.schedule(neighbor_, pkt, txtime + delay_);
                s.schedule(this, &intr_, txtime);
                busy_ = 1;
                busytime_ += txtime;
                if (trace_ != 0)
                        trace_->hop(src_, dst_, pkt);
        }
}
```

void Link::handle(Event* e)

This function verifies whether the link is busy or not, if it is, it marks it idle (not busy), and then it calles the send function to handle this event (as described above). Since the link class is the base class for all other link classes, so an object of this class can never get formed. Hence it implies that this particular handle function never gets called, so we can say that whenever this function is called it can only be through an interrupt.

void Link::send(Packet* pkt)

This is different from the previous send function as it takes the packet as an argument. This function accepts a packet for transmission on the link. The node class calls this method unlike the previous send function which was called by the handle funciton. The log-packet arrival and departure functions are called to update the statistics for the queue such as size in terms of bytes and also the number of packets and updates them and output them in a file. These functions also invoke the plot function if a Trace object is attached to it.

Link::command

This function is called from the interpreter to interpret Tcl commands. The following commands are interpreted here:

```
$link install $src $dst
$link stat $class $which
$link integral qsize
$link integral qlen
$link integral util
$link queue-trace $trace
```

This can execute all the operations that can be defined on the link. Since the link class inherits from the NsObject class, so this is virutal function that Link had to define on its own. So for the cases which Link cannot handle, it passes over (i.e. returns to) the NsObject class making it responsible for further handling. The commands having the above mentioned syntax are handled by this function. First the arguments are checked and then the strings are matched and appropriate action is taken. Thus you can gather statistics from your Tcl input by executing these commands.

Link::remove

This function is used to remove the link when it has been dropped or something.

Link::log_packet_arrival/departure

These functions are related to the statistics gathering functions of link. When a packet is put on a link the arrival statistics qnb etc need to be updated, similar is the case with departure statistics. But these functions are invoked only when the trace mode is on and the network communication traffic is being monitored.

Link::reset

This function is used to reset the link statistics that the user might be interested in. So that after reseting them and running the simulation, they can monitor the actual statistics gathered.

From the link various types of links are derived. These include the following

LossyLink (lossy.cc)

This inherits from the basic Link class. The only functions that are different are the enqueue and dequeue functions.

void LossyLink::enque(Packet* p)

The enqueue functions takes in the packet as an argument. It invokes the enqueue function and inserts the packet in the queue. It then logs the packet arrival. Then it invokes the loss_policy function which determines whether the packet is to be discarded or not. It then calls the remove function to remove the packet if the loss-policy allows for it. otherwise it lets it go.

Packet* p LossyLink::deque()

This simply dequeues the packet from the link when it is time for it to get off it and go to some other link, switch or node. It records its departure.

It can loose packets according to 2 different disciplines. Deterministic or uniform. This is implemented by the following function

virtual int packet_policy(Packet*) =0;

The classes **UniformLossyLink** and **DeterministicLossyLink** use their own methods for determining whether to discard the algorithm. In **UniformLossyLink** the loss depends according to a random uniform number. If the random number is more than the loss probability the packet is not lost.

```
UniformLossyLink::packet_policy(Packet *p)
{
        double val = Random::uniform();

        if (val > loss_prob_) {
                return (1);
        }
        return (0);
}
```

DropTailLink (drop-tail.h, drop-tail.cc)

This class inherits from the Link class. The functions that were declared as virtual in the Link class i.e. the enqueu and dequeue functions are defined separately for this DropTail Link class. It is identified by the DropTail Matcher class which inherits from the main Matcher class. This class is necessary when taking input from tcl script (i.e. the input file) since it verifies and matches the string and creates the new droptail link. So it is necessary when there is need to create a droptail link.

DropTailLink::enqueue(packet* p)

This function invokes the enqueue function of the queue class. It also checks to see if the total number of packets currently in the queue qnp in the queue is exceeding the limit qlim assigned to the queue or not. If it is exceeding the limit then, it is removed from the queue and also the log-packet departure statistics are also updated. (This is actually the drop-tail algorithm).

DropTailLink::dequeue()

This function is invoked when the packet is supposed to get off the link to get onto the destination node, the switch buffer or possibly another link. It simply dequeues the first packet in the queue that needs to be dequeued according to the time. It then hands it over to the log-packet-departure function which updates the statistics.

REDQueue (red.cc)

This type of a link implemented the RED algorithm also inherits from the main link class. It only has those functions that have been mentioned to be virutal in the link class. It also has its own reset function which overrides that of the basic class. For further details on this class it is necessary to understand the algorithm of which this is an implementation. So in order to gain knowledge of the algorithm refer to the following paper Sally Floyd and Van Jacobson " Random Early Detection Gateways for Congestion Avoidance" http://www-nrg.ee.lbl.gov/floyd/red.html

CBQueue (cbq.cc)

This is the implementation of the Class Based Queueing. Again the details of this algorithm are available at ftp://ftp.ee.lbl.gov/papers/params.ps.Z. The CBQ class uses packet-by-packet and round-robin queueing.

Node (node.h, node.cc)

This class **Node** inherits from the **NsObject** and **Hander** classes. The node represents the physical nodes of any network. They contain the following attributes

next_	The pointer to the next node in the network
cnt_	A static total number of nodes in the network.
addr_	The address of this node in the network. It is assigned by the value of cnt_
nport_	The number of agents attached to this node.
route_	The list of links. Used for routing. route_[destination] gives the link to route the packet
links_	The list of outgoing links from this node
maxport_	The maximum number of ports (agents) that can be attached to this node
demux_	The table of agents attached to this node. demux_[p->dport_] gives the required port
mdemux_	It is the list of multicast agents

Link State and Group Entity structures have not been done as yet. Since there is no next pointer in the link class, so in order to maintain the List of links attatched to a node. The Link State structure is the basic unit of this linked list of links.

This class inherits from the parent Matcher class. The matcher function takes input from the input file and creates the object.

Node::Node()

The Node constructor initializes the variables. It assigns the next count number as an address to the newly created node. It also initializes the maximum number of ports that is 64 (preset). This will double whenever this gets full. It also adds an entry in the table of maxagents and initializes them as well.

Node::~Node()

The destructor function deallocates the memory assigned to the routing table and the table containing the traffic descriptions.

Node::free(AgentList* p)

The free function is invoked when a traffic has completed its duration and is stopped. So this function deletes its entry from the list of Agents.

void Node::add_link(Link* p)

The add_link function is inserting the newly created link at the beginning of the linked list. It then makes an entry for this in the LinkState List.

void Node::setroute(int dst, Link* p)

The setroute function is invoked by the route-logic class. It returns the link on which the packet is supposed to be routed to if it comes destined for a certain destination.

int Node::add_agent(Agent* p)

The add_agent function adds the agent in the table of agents and returns the port number that it has assigned to the new agent. In case the number exceeds the total number of agents allowed (maxport) the array is doubled (maxport is also doubled) and then the very same procedure as above is applied.

send-graft, add-group,send-grafts, record-prune, send-prune are functions that facilitate multicasting of TCP in ns i.e. to send messages to a group (if required).

void Node::unicast_forward(Packet* p)

The unicast_forward function is required for normal transmission of packets as compared to the multicast_forward. It takes in the packet as an argument and checks whether it is destined for itself (i.e. the particular node that calls this function), if it is so it takes the destination port from the packet (an attribute of it) and assigns it to the port. Then it finds out the agent from the reference of port number. If there is an error condition i.e. the agent doesn't exist it is freed and appropriate actions are taken. If not, the packet is handed over to the receive function of the agent which takes appropriate action and the function returns. If not, it is routed to the appropriate link on which it is supposed to be routed given the destination of the packet (refering from the routing table). and then the packet is handed over to the send function of the link which handles it. This function is invoked by the handle function which is supposed to handle the event as required, either multicast forward or unicastforward.

void Node::handle(Event* e)

This function handles an event for this node. If the route does not exist then it exits. Else it determines the type of the packet, i.e. whether it is multicast or unicast and forwards it accordingly.

int Node::Command

This function is used to interpret the user input. It can be called as

```
$node agent $port
```

```
$node addr
```

In the first case it returns the address of this node. This is used when creating connections where the source or destination node's address has to be found. In the second case it returns the agent attached to the specified port.

RouteLogic (route.cc)

RouteLogicMatcher is the class that inherits from the Matcher class. This class identifies the route-logic object which is bring used. It sets the routing table of all the nodes to all other nodes based on the minimum hops routing logic. Whenver a new node is created the routelogic object is created and is assigned to the node and the routing table is built up once again. This implements the RouteLogic class that inherits from the TclObject class.

RouteLogic::command

This function can be used to either, call the routelogic object to compute the routes or for inserting a node to the routing logic's list of nodes.

RouteLogic::compute-routes()

This function actually implements the min-hops algorithm. i.e. by looking at the destination it calculates the minimum path from the particular node to the destination. Every time a node is added this function is invoked to update the routing table.

Agent (agent.h, agent.cc)

Agents are the objects that actually produce and consume packets. They are the transport entities and/or processes that run on end hosts. Each agent is automatically assigned a port number unique across all agents on a given node (analogous to a tcp or udp port). The agent knows the node with which it is connected to so that it can forward its packets to it. It also contains the packet size, the type of traffic, and the destination address (both node and agent). The Agent class is the base class for all types of TCP implementations. These include the TCP agents including Reno, Newreno, sack1 and also the TCP sinks including DelACKSink, SACK1TCPSink, SACK1DelACKTcpSink. There are corresponding matcher classes for all these agents. 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. The class agent inherits from the **Handler** and **NsObject** classes. This is the base **Agent** class, those classes inheriting from this class have to add the functions declared as virtual on their own. For examples all the classes that will inherit from the agent class will have to make their own destructor and receive functions. Some of its variable members are as follows:

node_	The node to which this agent is connected. This is needed for passing a packet generated by the agent to be handed over to the node for further transmission.
sport_	This is the port to which the agent is attached inside this node
daddr_	This is the address of the destination node.
seqno_	The current sequence number of the packet to be sent
size_	The size of the packet to be sent
type_	The type to be placed in the packet header
class_	The class to be placed in the packet header

Functions

Agent::Agent (int pkttype)

It sets the default values for different variables required to be set. It calls the memset function that sets the pending and sizeof pending values. It also invokes the link_int function it initializes the class variable and its values.

int Agent::command(int arg, const char*const* argv)

It can take in the following commands

- **$agent addr** - Returns the address of the node
- **$agent node** - Returns the name of the node
- **$agent dst-addr** - Returns the address of the destination

- **$agent port** - Returns the port of this agent
- **$agent dst-port** - Returns the port of the destination port
- **$agent seqno** - Returns the sequence number of the cells sent uptil now
- **$agent join $group** - Join the group
- **$agent leave $group** - Leave the group
- **$agent node $node** - Attach this agent to the node
- **$agent connect $node $port** - Connect this agent to the port of the node

void Agent::handle(Event* e)

The timer can handle two types of events, it can either be a packet to be handled or it can be a timer interrupt. If it is a packet, it sends it to another function **Agent::recv(p)** to handle. If the program is working well the packet should not come to the base class NullAgent, it ought to have been handled by the source or sink agents. This NullAgent only frees the packet. It only sets the Timeout timer since it is the base class.

Packet* Agent::allocpkt(int seqno) const

This function allocates a packet using the **Packet::alloc()** function. Then it initalizses all its generic attributes and returns the packet. It only takes in the sequence number of the packet as an argument. This function is invoked for example whenever a packet needs to be generated. e.g in the tcp.cc file void TcpAgent::output(int seqno) it calls this allocpkt function to get a packet with the given sequence number and all the parts of the header initialised.

The files named tcp-* contain the various implementations of agents.

Trace (trace.h, trace.cc)

This class inherits from the NsObject class. It is useful for outputs of the results of the simulation into an output file.

int Trace::command

It accepts the following commands.

- **$trace detach -** Closes the channel to the output file
- **$trace flush -** Flush the current channel
- **$trace attach $fileID -** This creates a channel for this trace object and attaches to the specified file
- **$trace callback $proc -** It resets the callback procedure name.

void Trace::format(int tt, int s, int d, Packet* p)

This function formats the output in the form of a string. The dump function eventually dumps it into the output file. This is a sample of the input file, test-suite.tcl which activates the trace. It opens the traceFile out.tr for writing. It is also given the time at which the trace is supposed to begin.

```
proc openTrace { stopTime testName } {
        exec rm -f out.tr temp.rands
```

```
        global r1 k1
        set traceFile [open out.tr w]
        ns at $stopTime \
                "close $traceFile ; finish $testName"
        set T [ns trace]
        $T attach $traceFile
        return $T
}
```

Other Helper Classes

Integrator

This class is used for accumulating the statistics of an object. It takes into account the length of time for which a certain entry remains in action. It is useful for finding the averages over time like the queue lengths etc.

Sigma

This class simply cumputes the mean of the data entries.

Random

This class generates uniform or exponential Random numbers which are the Random Early Detection algorithm.

ATM Classes

ATMNetwork (atmnetwork.h, atmnetwork.cc)

The AMTNetwork is an actual complete ATM Network. It has been derived from the Node class so that the TCP traffic can be simply routed to the Network. The Network appears to the TCP traffic as just a single node. The advantage is that we do not have to change the TCP routing logic and still perform independent routing inside the ATMNetwork. The Network knows all the ndoes and the links. The connection admission control and the routing logic is also implemented here. The user can specify his/her own routing logic and the connection Admission control algorithm. The number of nodes in the network is limited only by memory. The number of nodes simply double whenever all the nodes are used. The links are maintained as a matrix of links. When a packet comes to the network, it is broken down and handed over to the appropriate node. Later when the packet is recovered it is sent back to the network which passes it on just like any normal TCP node. There can be multiple ATM Networks declared in the total topology. This adds another dimension to modelling networks.

Variables

nodes	This is a list of all switches and sources counted as ATMNodes in the ATMNetwork. This can only contain objects of type ATMNode and not of other types.
map	This is used to identify the external TCP connections corresponding to the ATMNetwork connections. The map variable, when given a TCP node address returns an ATM Node index. map[tcpnode address]->ATMNode index
earliest_start	The earliest time a packet a packet can arrive from any link. This is needed to ensure that the packets from different links do not overlap. The overlapping can occur since the ATM Links can only transmit at slots and while a cell is waiting for a slot other packets can arrive.
curr_nodes	This gives the total number of nodes in the network at that particular moment.
max_nodes	This give the total number of nodes that can possibly be in the network, or the maximum number of nodes that can be declared. This is arranged such that whenever the current number of nodes is equal to the max_nodes, it is doubled, so that the network is not constrained by it. Hence the only limitation on increasing the total number of the nodes in the network is that of the memory.
link_table	his is a table of ATMLinks contained in the network. The links are stored in the form of the source and destination in the table. Hence each link has a unique source and destination. So given a destination and a source number it returns the corresponding link. link_table[src][dest]-> the link.
routelogic	This is gives the routing logic that is supposed to be followed by the ATM network. Route logic knows the topology of the whole network , it is atatched to the network, and the routing logic is implemented to the whole network.

Functions

static class ATMNetworkMatcher:public Matcher

This is the matcher function of the ATMNwork class. It inherits from the main Matcher class. This class identifies the ATMNetwork object by using the identifier **"atm-network"**. The **"new node atm-network"** is actually creating an object of type ATMNetwork.

int ATMNetwork::lookup_id(const char * id)

It is a function which when given the name of the object returns the index of the object with reference to the whole network. It looks up the name is located by using the name() function and comparing with all the values in the list of nodes. If it is found then the index of the object in the list of nodes is returned.

int ATMNetwork::attach(Node * t, ATMNode* a)

This function records that which TCP Node is attached to which ATM Node. It uses the TclObject::name() function to identify objects. If the function is successful in recording the name of the function it returns 1 else it returns a 0. The function actually looks in the map function (it gives it the address of the node) and it returns an index . The function then add the dummy link to the node and then returns.

int ATMNetwork::command(int argc, const char*const* argv)

This function is called from the Tcl input and by matching the user command performs the actions associated with it. Upon coming accross the "route" keyword it calls the routelogic object to Associate a routing logic with the network. If it is unable to find the object it returns false, otherwise it returns ok.

Upon the "add-node" command it looks up the node and adds it to the network. As a last alternative if the ATMNetwork is unable to handle this command it passes it over to the class previous in hierarchy (i.e. one above it), in this case (Node::command(argc, argv)).

void ATMNetwork::add_node(ATMNode * node)

This function is called through the command function upon seeing the add_node keyword. It actually adds the ATMNode to the network. If the nodes reach their maximum then simply double the array. i.e If the current number of nodes becomes equal to teh maximum number of nodes, the maximum number of nodes is doubled.and all the nodes are initialized.

Then the ATM adaptation layer corresponding to that particular report is initialized. Then the address of the node is set to be the current sequence number. The node is made to point to the network so that it knows the network to which it is associated with. At the same time the node is added to the list of nodes at the index of the current number of nodes. The route logic object is also initialized. It is made to the current number of nodes, the links and the routing table.

void ATMNetwork::handle(Event * e)

This function is responsible for handling the functions particularly generated for this network. If a packet comes it is broken and passed on to the appropriate ATM Node. When a packet is recovered the outgoing packet is handed over to the TCP network. When a packet enters the network for the first time its destination is pointing to its node. It is broken into 47 bytes but it is transmitted at a rate of 53 bytes to the atmnode (this is after the header is attatched. At the same time the destination is changed to some other destination rather than the current node which is initally the case. When this packet arrive again its destination is changed and it is treated like just another TCP packet. If the destination address of the packet is not equal to the current node it means this is a recovered packet which is about to leave the ATMNetwork. If this is the case then its original destination is restored it is signalled to the TCP network. If the destination address is the address of the current node then the packet is a new packet entering the packet for the first time. It is handled as follows.

Its local destination is changed to one more than the current address. This packet is broken into cells and its source and destination are altered as compared. The map function called for the source of the ATMCell gives the ATMNode from which this particular packet should be sent. The outgoing TCP node gives the ATM destination. The cell priority is set to 0 since all TCP traffic is considered to be of UBR (Unspecified bit rate). The flag of the cell that indicates whether the cell is queued or not is also set to 0. A new link which is the input ATM link is also created. The transmission time of the cell is calculated on the basis of 53 bytes (47 + the headers attatched). Routing tables are determined based on the routing strategy established. This also returns any delay associated with the establishing of the routes. If there are multiple packets arriving on the same link (i.e. implying queuing) the earliest_start function gives the time for which the packet must wait before it can be handled. The packet is broken into the number of cells it is supposed to be broken into then the next earliest start time is also calculated. After this the packet is handled as a normal TCP packet, using Multicast or unicast forward routing algorithms.

void ATMNetwork::add_link(ATMLink* link, int src, int dst)

This function adds teh link to the list o fall links in a network. A link can only have a unique pair of ends, i.e. there cannot be multiple links between two nodes. The link_table function returns the particular link if given a source and a destination.

ATMAal (atmaal.h, atmaal.cc)

In real life the AAL layer needs to keep record of all the cells in a particular packet from a connection. In our design we only need the last cell received. This is valid since the aal layer is separate for every source. So from one source cells from two different packets will arrive in the same sequence i.e. the cells from the two packets would not overlap. Thus only looking at the next cell, if it is our of sequence we can empty the aal buffer. Also cells are dropped here if they are out of sequence indicating that the packet is incomplete.

Variables

flag_	This is the buffer that is used to indicate the state of the buffer. It shows a 0 if the buffer is empty and a 1 if the buffer is full.
seq_no_	This is the private variable that shows the sequence number of the cell, and keeps record of it in the ATM Adaptation layer.
tcp_seq_	TCP packet sequence number
tcp_port	The sequence number of the packet and its port number identify whether the cells belong to the same packet and connection or not. Since the port number is unique to a particular connection. We need the port number to identify the cells of the packet because two packets from different agents i.e. (different port numbers) can have the same sequence number.
tcp_addr	The address of the particular node with respect to the topology of the network. This is necessary alongwith the tcp_port_ number of two cells for them to be in sequence.

Functions

ATMAal::ATMAal()

This is the constructor, which initializes all the above mentioned attributes of the ATM adaptation layer.

int ATMAal::Receive_Cell(ATMCell* cell)

This is the main function of the ATM Adaptation layer to reassemble the cells coming in and converting them into TCP/IP packets. This function receives a cell and returns whether a packet is received or not, if it is then it returns a 0 else it returns a 1. The function checks if the cell received has sequence number 1, if it is so it means it is the first cell of any pacekt, so it inserts the cell into the buffer and marks the buffer full. If it is theonly cell of the pacekt then return full. Else wait for other sequence numbers of the same packet.

If the bufferis empty i.e. the flag is = 0, and the packet is not of sequence number 1 then it is supposed to be discarded, since the AAL layer is not supposed to recover the pacekt it is just supposed to mark it and discard it.

Now if the cell that comes is the next one in sequence then it is inserted. If again this is the last sequence number of the cell then return full and the buffer is marked empty. Else the expected sequence number is advanced and the buffer is marked full.

ATMNode

This is the parent class for all the nodes in the network. This includes the ATM traffic sources and the switches. It is derived from the NsObject and Handler classes so that they can handle packets and also accept tcl commands. We kept it seperate from the TCP node because we wanted to keep them excluded from the TCP routing algorithm. The nodes include both ATM Switches and ATM Sources. The nodes know the network they belong to and the links that are attached to them. They also contain dummy links, which are just used to identify the external links by which these nodes are implicitly connected. The ATMNetwork is the actual entity that is connected to the TCP nodes. Similarly they have their own routing tables. The ATM AAL layer is also implemented here.

aal	This is conceptually how the ATM Adaptation layer is implemented. Each node has an AAL layer for each node in the ATMNetwork. This works because two packets from the same node in the network can not overlap. i.e. they will arrive in sequence.
total_nodes	This gives the total number of nodes in the network.
no_links_	This gives the total number of links connected in the network.
addr_	This gives the address of the node.
network_	This serves two purposes. Firstly the node should know which network it is pointing to. Since after a packet is complete it is handed over to this network. This network handles the packet. Also the node is supposed to refer to the network when it is in need for resources.
links_	The list of outgoing links from the node. These are all the atm links emerging from the node, this does not include any other links
dummy_	This is a list of all the dummy links (external links). The dummy links are It has two attributes to it. One is the address of the external node, and the other is the port number of the switch. Each individual component also contains a pointer to the next link. This is a component of the linked list which contains the dummy lists. The need for the dummy link arises since the external links are physically connected with the ATMNetwork and not with the individual atmnode.
dummy_count	This gives the total number of dummy nodes attatched to the node
port_table	The list of ports in this switch.

Functions

ATMNode::ATMNode()

This is the ATMNode constructor. The number of ATMLinks is initially 16 and doubles whenever the capacity reaches its maximum. The port table is also created for 16 entries corresponding to the number of ATMLinks.

ATMNode::ATMNode(int tn)

This is another constructor which changes the default value of the total number of nodes equal to the number given to it as argument.

void ATMNode::init_aal(int tn)

This function initializes the aal layer. This function is invoked by the ATMNetwork whenever there is a change in the total number of nodes in the network. This function initializes the total_nodes equal to the number given as argument. Then creates an object of type ATMAal of size equal to the total number of nodes.

void ATMNode::add_dummy_link(int ad)

This function adds a dummy link to this node for the specified address given to it as an argument. This function is invoked by the network when it connects a tcpnode to an atmnode.

void ATMNode::get_dummy_port(int ad)

This function searches for the address in the list of dummy ports and returns the port number of the queue. This function gets the address of the node to be searched. It them compares and searches for it in teh list of nodes. It returns a -1 if it does not find the required port not. if it does then it returns the port number of the queue.

void ATMNode::add_link(int ad)

This function adds a link to the particular node for which it is invoked. If the total number of nodes in the atm network reaches the maximum number that there can be, the maximum number in the network is doubled. It updates the network since a new node is being added to the network. It also makes a new entry into the list of links. Apart from this the port_table is also increased. The total number of links is also incremented.

ATMCommand (atmcmd.cc)

This is inherited from the TclObject class. User input command is translated into the command function by identifying the keyword "connect".

int ATMCommand::command(int argc, const char*const* argv)

This handles the command for connecting a tcp node to an atmnode e.g, it responds to user input of the following sort in the test-suite.tcl file. It searches for the tcpnode, the atm network and the atmndoe and records in **map** array of the network which node is connected with which internal node. Also it creates a dummy link in the atmnode.

ns connect $tcpnode $atmnetwork $atmnode

ATMLink (atmlink.h, atmlink.cc)

The atmlink inherits from the base link class. The difference between this class and all the other classes is that the end nodes in this case are the atm nodes, and also the transmission time of the cells from end to end is fixed. The transmission time is fixed due to the standard characteristic that the cell size of the ATM cell is fixed i.e. it is 53 bytes. The atmlinks are slotted in nature they are synchronized and start at the same time i.e. 0.0, so a cell can only get on a link if its time is a multiple of the size of the slot.

neighbor_	This identifies the end node for the particular link.. It is needed so that any cell arriving on the link can be routed to this node.
start	This attribute gives the start time at which a cell can get on a particular link. If the time at which teh cell is supposed to get on the link is not a multiple of the slot size, it is adjusted so that it becomes one and meets the criterion. This is needed for synchronizing the arrival of cells on exact slots.
available_bw	The bandwidth available on the link. It is used by the Connection Admission Control.

Functions

static class ATMLinkMatcher

This class inherits from the base matcher class. It is used for identifying the ATMLink. The type "**atm"** identifies it to be an ATMLink. It matches with the input and creates a matcher_atm_link;

ATMLink::ATMLink()

This is the constructor for the ATMLink class. It initializes the private members of the ATMLink class the neighbour and the starting time of the links to 0.0.

inline void ATMLink::neighbour(ATMNode* p)

This constructor sets the neighbouring to the ATMNode p passed to it as a argument. It also initializes the destination address to the address of the node given to it as an argument. This too is a sort of a constructor.

void ATMLink::handle(Event* e)

This is the handle function for the ATMLink class. It handles teh event that is passed to it. Its main function is to find the slot size. Then it schedules the time at which the cell is supposed to get off the link and reach the other end. Since there is a restriction on slotted ATMLinks that cells can only get on and off the links on a given slot. The slotsize is calculated by dividing the total number of bits in a cell i.e. (53* 8 = 424) by the bandwidth of the link which is in bytes per second so this gives the slot size in seconds. Then an instance of the Scheduler is created. The current time cur_time is set equal to the time of the scheduler. If this curr_time is less than the start time of the link, then current time is assigned equal to the start time. Then the slot size is added to the current time. If time lies between two slots it is upgraded so that it falls on the next slot. Then the neigbour, the cell itself, the next slot time and the delay is passed to the schedule function of the newly created scheduler object. The slotsize is added to the time for which the link will remain busy.

int ATMLink::command(int argc, const char*const* argv)

This function is used to override the "install" command to be between two ATMNodes. It checks that both the ends are atmnodes, if the source and destination are the same then there is an error. If not then the destination is made the neighbour. The link is then added to the source. Otherwise the command is returned to the command function in the above hierarchy.

ATMRouteLogic (atmroute.h, atmroute.cc)

This is the class that inherits from the main TclObject class. This class establishes the route from all sources to all destinations. The routing logic can be used any which way it is required as.

nodes_	This gives the total number of nodes in the network.
curr_nodes_	This gives the total number of nodes currently there in the network.
lnk_tbl_	This is a link_table which when given a source and a destination, returns a link.

class ATMMinHop: public ATMRotueLogic

This class uses the minimum hop algorithm between all sources and all destinations. At the very first invokation the algorithm tries to establish paths from every node to every other node i.e. all to all paths. If a node is added subsequently i.e. after adding the routes are computed, the done is set to 0 and the routes are established once again.

class ATMMinHopRouteLogicMatcher: pubic Matcher

This class inherits from the matcher clss. This class class identifies the match with the identification "min-hop". It matches this keyword at the ns_default.tcl file and matches it appropriately.

int ATMMinHop::command(int argc, const char*const* argv)

This matches the command at the user input and invokes the appropriate function. In this case, if it comes accross a "compute-routes" keyword it invokes compute_routes() function.

The other keyword that it matches is "insert" it inserts the node that is added to the network topology.

int ATMMinHop::establish_route(int src, int dst)

This function creates an all to all shortest paths and returns 0 as the delay for time taken in establishing the routing tables.

ATMSource (atmsrc.h, atmsrc.cc)

The class ATMSource inherits from the main ATMNode class. ATM Agents are attached to this which are the actual generators of the traffic.

static class ATMSourceMatcher:public Matcher

This is the matcher class for the ATM source. It simply matches the object atm-source declared by the tcl input. Upon matching the keyword it returns the object ATMSource.

void ATMSource::handle(Event* x)

This function is supposed to handle the events meant for the ATMSource to handle. If the node for which the function gets invoked is the destination node as well then the event is handed over to the appropriate Agent. If the event is the generation of a cell then the next cell is also generated.

ATMSwitch (atmswitch.h, atmswitch.cc)

class ATMSwitchBuffer

This class is used to implement the buffer for the switch. Different switching architectures can be implemented by making them inherit from the main buffer class. We are not required to store the actual cells, only the total count of the cells is fine since it serves the purpose, and saves the memory requirements as well.

Variables

qsize	gives the current size of the queue
total_q	gives the sum of all the queues
ports	This gives the total number of ports.
qlimit_	This is the limit assigned to the queue which is used to compare against in deciding which packet to discard and which one to let go.
exit_time	The time at which the last cell will leave the queue. The next cell exit time is this time plus one cell time if the exit time has not yet arrived or the current time. Otherwise it is the current time plus the cell transmission time.

ATMSwitch

This is the generic (base) class for all ATMSwitches. All the subsequent classes of switches i.e. the output buffer and shared memory switches inherit from this class after slight variation in the functions.

Variables

buffer_size_	This gives the buffer size of the switch in terms of number of cells.
qlimit	This gives the threshold value against which the discard decision is taken or ruled out.
no_of_cells_lost	This gives the total number of cells lost in the buffer of the switch.
no_of_cells_passed	This gives the total number of cells passed through the switch buffer.

Functions

ATMSwitch::ATMSwitch()

This is the constructor for the ATMSwitch class. It declares a new buffer of the type ATMSwitchBuffer. It also initialises the link_int with the name of the switch ns_switch, the buffer-size, and

ATMSwitch::add_link(ATMLink* link)

This function adds a link to the switch. It also updates the buffer size allocated to each port on the addition of the new link. This also sets the buffer limit which is used in implementing discard decisions.

ATMSwitch::command(ATMLink* link)

This function interprets the user input from the ns_default.tcl file and interprets it and takes appropriate action. This command function reacts to one of the following commands in the ns_default.tcl files. e.g

```
$switch discard-algo $discard
```

On getting the keyword discard-algo, looks up the name in the list of algorithms and sets the appropriate discard algorithm. The return value of the function is assigned to the discard variable.

```
$switch buffer-size $size
```

This line is interpreted so as to set the buffer size. If none of the options are matching, then this function returns to the command function in the class one step above in object hierarchy.i.e. the command function of the NsObject::command.

ATMSwitch::handle(Event* x)

The main purpose of this function is to handle the event passed to it as an argument. If the node for which this function is invoked is the destination and the cell is currently queued up then the event is passed on to the appropriate ATM adaptation layer. If the Aal Layer informs that the packet is complete then the packet is passed back to the network.

If the current node is the destination but the cell is not queued then the cell is to be queued with the with the appropriate dummy port. It is enqueued

If the current node is the destination as well as the source then the packet is returned.

If this is not the destination and the cell is not queued then enque the cell and update the statistics. else deque the cell and update the statistics.

void ATMSwitch::enque(ATMCell* cell, int port)

This takes in a cell and enques it in the appropriate port. It also updates the queue lengths of the buffer and the statistics of the switch.

void ATMSwitch::deque(ATMCell* cell, int port)

This takes in a cell and deques it (changes cell->queued_ to 0) from the appropriate port. It also updates the queue lengths of the buffer and the statistics of the switch.

Types of Switches.

Two types are switches are being implemented in our model of the atm-layer. The output Buffer switch and the Shared memory switch.

ATMOutputBuffer

The output buffer switch inherits from the main ATMSwitch class. It is just a simple switch but the buffer limits have been set differently. The traffic from different connections gets queued up in different buffers at the output port.

ATMSharedBuffer

The shared memory switch also inherits from the main ATMSwitch, but it implements logical queues which share the same buffer space. When a particular connection is closed its queue vanishes from the memory of the queue.

This class is used for identifying the switches. It operates on swwing the following sort of commands at the test-suite.tcl file. The above commands are used in the input file when creating the specifics for the atmnetwork. This command

set n1 [ns node atm-switch output early] set n2 [ns node atm-switch shared simple]

The set n1 command creates an output buffer switch, which implements early packet discard. On the other hand the set n2 command sets the node with the at-switch which is shared memory and implements the simple packet discard algorithm. Upon finding the atm-switch keyword, it matches whether the type of the switch is there or not, then it returns the particular switch with the specified packet discard algorithm implemented at the buffer.

void ATMOutputBuffer::set_buffer_limit

The buffer_size is equally divided accross both the atmlinks and the dummy external links.

void ATMOutputBuffer::set_buffer_limit

The buffer size is the queue limit. The buffer limit in the shared memory switch is the size of the buffer since the buffer is shared by all the connections.

ATMDiscardAlgorithm

The Discard algortihm is implemented in such a way that the user has to write just one function and the algorithm is implemented. It has one function which is supplied the switch buffer the current cell and the port to which is supposed to be routed to. It can then decide whether to discard the packet or not. For the integrity of the switch, the discard algorithm can not change the buffer or the cell. Also it has to be supplied the port number because the discard algorithm does not need to know how and where to route the cell.

The simple output and shared buffer switch classes inherit from the DiscardAlgorithm class. These implement the Simple discard algorithm on the output and shared memory switches.

The early output and shared memory switch classes ineherit from the Discard Algorithm class. These implement the Early Packet discard algorithm on the Shared memory and output buffer switches.

class DiscardAlgorithm: public NsObject

This class inherits from the NsObject class. This is the generic DiscardAlgorithm class. All the discard algorithms can be implemented by inheriting from this generic class. The only difference between the generic DiscardAlgorithm class and those inheriting from it is that the only functions that determine (by looking at the buffer and the cell) whether the cell is to be discarded or not. which is the check_discard(const ATMSwitchBuffer* , const ATMCell*, int).

static class ATMOutputDiscardMatcher

This matcher class inherits from the base Matcher class. It is used to match the user input. These are particularly used to identify the discard algorithm. "output" specifies that it is for output buffer and "simple" and "early" identify the discard algorithm.

static class ATMSharedDiscardMatcher

This does the same thing as the previous function but for the Shared Memory switch instead of the Output buffer switch.

int SimpleOutputBuffer::check_discard

This function implements the Simple discard algorithm at the Output Buffer switch. When the individual queue of a port (connection) is full, it starts discarding the packets of that particular connection. The function returns a 1 if the packet is supposed to be discarded and a 0 if it is not to be discarded.

```
int SimpleOutputBuffer::check_discard(const ATMSwitchBuffer* buffer,
                                        const ATMCell* cell, int port)
{
        if(buffer->qsize_[port] >= buffer->qlimit_)
                return 1;
        return 0;
}
```

int SimpleSharedBuffer::check_discard

This function implements the Simple discard algorithm at the Shared Memory switch. When the combined total of the port queues is full. The function returns a 1 if the packet is supposed to be discarded and a 0 if it is not to be discarded.

```
int SimpleSharedBuffer::check_discard(const ATMSwitchBuffer* buffer,
                                        const ATMCell* cell, int port)
{
        if(buffer->total_q_ >= buffer->qlimit_)
                return 1;
        return 0;

}
```

EarlyOutputBuffer

weight is the percentage of queue to be filled before discarding the UBR cells.

int EarlyOutputBuffer::check_discard

If the percentage weight of the queue is full then the incoming UBR cells are discarded. But if all the queue is full then all the cells from all the connections are discarded. But the queue is examined at the specified port given as argument.

```
int EarlyOutputBuffer::check_discard(const ATMSwitchBuffer* buffer,
                                        const ATMCell* cell, int port)
{
        if(buffer->qsize_[port] >= buffer->qlimit_)
                return 1;
        if(cell->priority_)             //Not UBR
                return 0;
        if(buffer->qsize_[port] >= weight_*buffer->qlimit_)
                return 1;
        return 0;
}
```

int EarlySharedBuffer::check_discard

If a percentage (weight) of the queue is full then discard the incoming UBR cells. ELse if all the queue is full then discard all the cells. The difference between this type of discard and that implemented on the Output Buffer is that in this type of switch the whole buffer is examined rather than a part of the buffer as in the previous one. This function returns a 0 or a 1 depending on whether the cell is to be discarded or not.

```
int EarlySharedBuffer::check_discard(const ATMSwitchBuffer* buffer,
                        const ATMCell* cell, int port)
{
        if(buffer->total_q_ >= buffer->qlimit_)
                return 1;
        if(cell->priority_)             //Not UBR
                return 0;
        if(buffer->total_q_ >= weight_*buffer->qlimit_)
                return 1;
        return 0;
}
```

NS Defaults (ns_default.tcl)

The default values for most variables are specified in the ns_default.tcl file. If you have to change any of the defaults you can change them in this file. However if you do this you will have to recompile the program. A collection of such configuration parameters that can be modified as above either before a simulation begins, or dynamically, while the simulation is in progress. If a parameter is not explicitly set, it defaults to a value stored in a global Tcl array. These variables are stored in objects belonging to class **Var** in the static list of the NsObject Class. The following is the list of parameters that can be set in the program.

TCP Variables

These variables control the behaviour fo the TCP traffic. It controls the window sizes, packet sizes and various other TCP traffic related aprameters.

```
set ns_tcp(maxburst) 0
set ns_tcp(maxcwnd) 0
set ns_tcp(window) 20
set ns_tcp(window-init) 1
set ns_tcp(window-option) 1
set ns_tcp(window-constant) 4
set ns_tcp(window-thresh) 0.002
set ns_tcp(overhead) 0
set ns_tcp(ecn) 0
set ns_tcp(packet-size) 1000
set ns_tcp(bug-fix) true
set ns_tcp(tcp-tick) 0.1
set ns_tcpnewreno(changes) 0
```

and this is how these variables are accessed

```
TcpAgent::TcpAgent() : Agent(PT_TCP), ds_(0), rtt_active_(0), rtt_seq_(-1)
{
        // read/write links
        link_real("ns_tcp", "window", &wnd_, 0);
        link_real("ns_tcp", "window-init", &wnd_init_, 0);
        link_int("ns_tcp", "window-option", &wnd_option_, 0);
        link_real("ns_tcp", "window-constant", &wnd_const_, 0);
        link_real("ns_tcp", "window-thresh", &wnd_th_, 0);
        link_real("ns_tcp", "overhead", &overhead_, 0);
        link_real("ns_tcp", "tcp-tick", &tcp_tick_, 0);
        link_int("ns_tcp", "ecn", &ecn_, 0);
        link_int("ns_tcp", "packet-size", &size_, 0);
        link_bool("ns_tcp", "bug-fix", &bug_fix_, 0);
        link_int("ns_tcp", "maxburst", &maxburst_, 0);
        link_int("ns_tcp", "maxcwnd", &maxcwnd_, 0);

        // read-only links
        link_int(0, "dupacks", &dupacks_, 1);
        link_int(0, "seqno", &curseq_, 1);
        link_int(0, "ack", &highest_ack_, 1);
        link_real(0, "cwnd", &cwnd_, 1);
        link_real(0, "awnd", &awnd_, 1);
        link_int(0, "ssthresh", &ssthresh_, 1);
        link_int(0, "rtt", &t_rtt_, 1);
        link_int(0, "srtt", &t_srtt_, 1);
        link_int(0, "rttvar", &t_rttvar_, 1);
        link_int(0, "backoff", &t_backoff_, 1);
}
```

RED parameters

These variables control the behaviour fo the RED (Random Early Detection) Gateways. For a detailed description of these parameters you should read the following paper Sally Floyd and Van Jacobson " Random Early Detection Gateways for Congestion Avoidance"
http://www-nrg.ee.lbl.gov/floyd/red.html

```
set ns_red(bytes) false
set ns_red(thresh) 5
set ns_red(maxthresh) 15
set ns_red(mean_pktsize) 500
```

```
set ns_red(q_weight) 0.002
set ns_red(wait) true
set ns_red(linterm) 50
set ns_red(setbit) false
set ns_red(drop-tail) false
set ns_red(doubleq) false
set ns_red(dqthresh) 50
set ns_red(subclasses) 1
set ns_red(thresh1) 5
set ns_red(maxthresh1) 15
set ns_red(mean_pktsize1) 500
```

and this is how thyey are initialised

```
REDQueue::REDQueue()
{
        Tcl& tcl = Tcl::instance();

        link_bool("ns_red", "bytes", &edp_.bytes, 0);
        link_real("ns_red", "thresh", &edp_.th_min, 0);
        link_real("ns_red", "maxthresh", &edp_.th_max, 0);
        link_int("ns_red", "mean_pktsize", &edp_.mean_pktsize, 0);
        link_real("ns_red", "q_weight", &edp_.q_w, 0);
        link_bool("ns_red", "wait", &edp_.wait, 0);
        link_real("ns_red", "linterm", &edp_.max_p_inv, 0);
        link_bool("ns_red", "setbit", &edp_.setbit, 0);
        link_bool("ns_red", "drop-tail", &drop_tail_, 0);

        link_bool("ns_red", "doubleq", &doubleq_, 0);
        link_int("ns_red", "dqthresh", &dqthresh_, 0);
}
```

CBQ based parameters

These variables control the behaviour fo the RED (Random Early Detection) Gateways. For a detailed description of these parameters you should read the paper by Sally Floyd at ftp://ftp.ee.lbl.gov/papers/params.ps.Z. The CBQ class uses packet-by-packet and round-robin queueing.

```
set ns_cbq(algorithm) 0
set ns_cbq(max-pktsize) 1024
set ns_class(priority) 0
set ns_class(depth) 0
set ns_class(allotment) 0.0
set ns_class(maxidle) 4ms
set ns_class(minidle) -0.2ms
set ns_class(extradelay) 0
set ns_class(plot) false
# set ns_class(qdisc) red
set ns_class(qdisc) drop-tail
```

Parameters for various Agents

For different types of agents different parameters can be set. These are assigned to different classes and stored in the Var list.

```
set ns_sink(packet-size) 40
set ns_delsink(interval) 100ms
set ns_sacksink(max-sack-blocks) 3
```

```
set ns_cbr(interval) 3.75ms
set ns_cbr(random) 0
set ns_cbr(packet-size) 210
set ns_rlm(interval) 3.75ms
set ns_rlm(packet-size) 210

set ns_ivs(S) 1
set ns_ivs(R) 1
set ns_ivs(state) 0
set ns_ivs(rttShift) 0
set ns_ivs(keyShift) 0
set ns_ivs(key) 0
set ns_ivs(maxrtt) 0.0
set ns_ivs(ignoreR) 0

set ns_source(maxpkts) 0
set ns_telnet(interval) 1000ms
set ns_bursty(interval) 0
set ns_bursty(burst-size) 2
set ns_message(packet-size) 40

set ns_facktcp(ss-div4) 1
set ns_facktcp(rampdown) 1
```

This is how these variables are set in the constructor.

```
FackTcpAgent::FackTcpAgent() : rampdown_(0), ss_div4_(0), retran_data_(0), fack_
(-1), wintrim_(0), wintrimmult_(.5)
{
        link_int("ns_facktcp", "ss-div4", &ss_div4_, 0);
        link_int("ns_facktcp", "rampdown", &rampdown_, 0);
}
```

Default values for links

For links of various types these are the default values that are set in via the constructor of the classes.

set ns_link(bandwidth) 1.5Mb
set ns_link(delay) 100ms
set ns_link(queue-limit) 50
set ns_lossy_uniform(loss_prob) 0.00

and this is how they are initiaslised

```
Link::Link() : busy_(0), neighbor_(0), qnp_(0), qnb_(0), qtrace_(0),
    flowmgr_(0)
{
        Tcl& tcl = Tcl::instance();
        link_bw("ns_link", "bandwidth", &bandwidth_, 0);
        link_time("ns_link", "delay", &delay_, 0);
        link_int("ns_link", "queue-limit", &qlim_, 0);

        link_int(0, "queue-length", &qnp_, 1);
        link_int(0, "queue-size", &qnb_, 1);

        ncs_ = 10;
        cs_ = new classStat[ncs_];
```

```
}
```

Default Values for Switches

```
set ns_switch(buffer-size) 40
set ns_early_discard(weight) 0.8
```

Functions

The ns_default.tcl contains several helper functions for the ease of the user for defining several. These functions rely on the **procedure ns** defined later and the **command** functions of every class. These functions include **ns_connect** which connects a source and sink agent. together. This is used by the **ns_create_connection** procedure.

```
proc ns_connect { src sink } {
        $src connect [$sink addr] [$sink port]
        $sink connect [$src addr] [$src port]
}
```

The **ns_create_connection** procedure takes in the type of source and sinks and the source and destination nodes and returns the source agent. It creates two agents of the specified type and connects them using **ns_connect** procedure.

```
proc ns_create_connection { srcType srcNode sinkType sinkNode class } {
        set src [ns agent $srcType $srcNode]
        set sink [ns agent $sinkType $sinkNode]
        $src set class $class
        $sink set class $class
        ns_connect $src $sink
        return $src
}
```

The **ns_duplex** procedure is used to create two links between two nodes of a certain bandwidth, delay, and of a certain type.

```
proc ns_duplex { n1 n2 bw delay type } {
        set link0 [ns link $n1 $n2 $type]
        $link0 set bandwidth $bw
        $link0 set delay $delay
        set link1 [ns link $n2 $n1 $type]
        $link1 set bandwidth $bw
        $link1 set delay $delay
        return "$link0 $link1"
}
```

This function creates a Reno TCP connection

```
proc ns_create_reno { tcpSrc tcpDst window start class} {
        set tcp [ns_create_connection tcp-reno $tcpSrc tcp-sink $tcpDst $class]
        $tcp set window $window
        set ftp [$tcp source ftp]
        ns at $start "$ftp start"
        return $tcp
}
```

This function creates a TCP CBR connection

```
proc ns_create_cbr { srcNode sinkNode pktSize interval class } {
        set src [ns agent cbr $srcNode]
        set sink [ns agent loss-monitor $sinkNode]
        $src set interval $interval
        $src set packet-size $pktSize
        $src set class $class
        ns_connect $src $sink
        return $src
}
```

This function Creates a CBQ based connection

```
proc ns_create_class { parent borrow allot maxIdle minIdle priority depth extraDelay
        global ns_class ns_link
        set class [new class]
        set class1 [ns_class_params $class $parent $borrow $allot $maxIdle \
          $minIdle $priority $depth $extraDelay 0]
        set qdisc [new link $ns_class(qdisc)]
        $class1 qdisc $qdisc
        return $class1
}
```

Create a class1 connection

```
proc ns_create_class1 { parent borrow allot maxIdle minIdle priority depth extra
Delay Mbps} {
        global ns_class ns_link
        set class [new class]
        set class1 [ns_class_params $class $parent $borrow $allot $maxIdle \
           $minIdle $priority $depth $extraDelay $Mbps]
        set qdisc [new link $ns_class(qdisc)]
        $class1 qdisc $qdisc
        return $class1
}
```

Set class parameters

```
proc ns_class_params { class parent borrow allot maxIdle minIdle priority depth
extraDelay Mbps} {
        $class parent $parent
        $class borrow $borrow
        $class set allotment $allot
        $class set maxidle $maxIdle
        $class set minidle $minIdle
        $class set priority $priority
        $class set depth $depth
        $class set extradelay $extraDelay
        set class1 [ns_class_maxIdle $class $allot $maxIdle $priority $Mbps]
        set class2 [ns_class_minIdle $class $allot $minIdle $Mbps]
        return $class2
}
```

procedure NS

This procedure is the heart of the input for our simulator. It calls the appropriate command functions depending on the object.

ns connect $tcpnode $atmnetwork $atmnode

This command is used to connect a tcpnode to the atmnode in the atmnetwork. This is handled by the class **ATMCommand** which looks up the three arguments and calls the **attach** method of the ATMNetwork.

```
if { $cmd == "connect" } {
        if { [llength $args] == 3 } {
                connect [lindex $args 0] [lindex $args 1] [lindex $args 2]
                return
        }
        else {
                puts stderr "ns: Syntax 'ns $cmd tcp_node atm_network atm_node'"
        }
}
```

The **node** commands handle all types of nodes in the network. This includes routers, TCP nodes, ATMNetworks, ATMSwitches and ATM Sources.

```
if { $cmd == "node" }
```

ns node atm-switch $type $discard-algo

For creating ATM Switches first a new node with the specified type of the switch is created. Then a new object of the type of the discard algorithm is created and attached to the switch. The **new command** is actually calling the **command** function of the class **CreateCommand**. The node is not inserted in the main list of nodes nor in the routing logic of the TCP. This keeps the routing logic of ATM independent of the TCP above and vice versa.

```
        if { [llength $args] == 3 } {
                if { [lindex $args 0] == "atm-switch" } {
                        set type [lindex $args 1]
                        set node [new atm-switch $type]
                        set algo [new $type [lindex $args 2]]
                        $node discard-algo $algo
                        return $node
                }
         }
```

ns node atm-network

ns node atm-network $route-algo

An ATM Network is just a standard node with its own routing logic. If the routing algorithm is not specified **min-hop** is used. Otherwise a new object of type route-logic is created and attached to the node. The network is inserted in the TCP routing logic and is visible as only a single node.

```
         if { [llength $args] == 2 } {
                if { [lindex $args 0] == "atm-network" } {
                        set node [new atm-network]
                        set routelogic [new [lindex $args 1]]
                        $node route $routelogic
                }
         }
```

```
 if { [llength $args] == 1 } {
        if { [lindex $args 0] == "atm-network" } {
                set node [new atm-network]
                set routelogic [new min-hop]
                $node route $routelogic
        }
}
```

ns node atm-source

This creates an ATM Source. ATM agents can then be attached to this node.

```
if { [lindex $args 0] == "atm-source" } {
        set node [new atm-source]
        return $node
}
```

ns node

This creates a TCP Node which can be either a router or a Tcp node. It is inserted in the main routing logic of the network.

```
if { [llength $args] == 0 } {
        set node [new node]
}
if ![info exists ns_compat(route-logic)] {
        set ns_compat(route-logic) [new route-logic]
}
$ns_compat(route-logic) insert $node
return $node
```

ns link $n1 $n2 $type

This command creates a link between the two nodes of the specified type. The type matches the corresponding Matcher class and the object created is of that type. If it is not an atm link then it also has to be inserted in the ns_compat.

```
if { $cmd == "link" } {
        if { [llength $args] == 3 } {
                set src [lindex $args 0]
                set dst [lindex $args 1]
                set type [lindex $args 2]
                set L [new link $type]
                $L install $src $dst
                if { $type != "atm" } {
                        set ns_compat(link:$src:$dst) $L
                }
                return $L
        }
```

ns link $n1 $n2

This command can be used to return the link that is connected with the two nodes. It calls the command function of the Link class.

```
        if { [llength $args] == 2 } {
```

```
                set src [lindex $args 0]
                set dst [lindex $args 1]
                return $ns_compat(link:$src:$dst)
        }
```

ns link

This command returns a complete array of all the links. This is useful for setting some parameters of all the links of the network.

```
        if { $args == "" } {
                set L ""
                foreach v [array names ns_compat] {
                        if [string match link:* $v] {
                                lappend L $ns_compat($v)
                        }
                }
                return $L
        }
```

ns agent $type $node

This command can be used to make agents of any types. They are attached to the nodes and returned.

```
if { $cmd == "agent" } {
        if { [llength $args] == 2 } {
                set type [lindex $args 0]
                set node [lindex $args 1]
                set agent [new agent $type]
                $agent node $node
                return $agent
        }
}
```

ns trace

This creates a Trace object and returns it. This can be attached to any node.

```
if { $cmd == "trace" } {
        set trace [new trace]
        return $trace
}
```

ns at "valid string"

This can be used to execute any valid Tcl command at the specified time. It calls the ATCommand class which handles it.

```
if { $cmd == "at" } {
        eval ns-at $args
        return
}
```

ns now

This command calls the command function of the NowCommand class. and returns the current time. It is useful for dumping statistics at various time intervals.

```
if { $cmd == "now" } {
        return [ns-now]
}
```

ns random

ns random $seed

This generates a random number.

```
if { $cmd == "random" } {
        return [eval ns-random $args]
}
```

ns run

This actually runs the simulator. It calls the command function of the RunCommand. It first computes the route logic in the network.

```
if { $cmd == "run" } {
        $ns_compat(route-logic) compute-routes
        ns-run
        return
}
```