APPENDICES

APPENDIX-A

Nodes and Packet Forwarding

This chapter describes aspects of creating a topology in ns, i.e., creating the nodes. Each simulation requires a single instance of the class Simulator to control and operate that simulation. The class provides instance procedures to create and manage the topology, and internally stores references to each element of the topology.

Node Basics

The basic primitive for creating a node is

```
set ns [new Simulator]
```

$ns node

The typical structure of a node is as shown in Fig A.1. This simple structure consists of two TclObjects: an address classifier (classifier_) and a port classifier (dmux_). The function of these classifiers is to distribute incoming packets to the correct agent or outgoing link.

![Fig. A.1 Structure of a Node](image)
The Classifier

A classifier provides a way to match a packet against some logical criteria and retrieve a reference to no other simulation object based on the match results. Each classifier contains a table of simulation objects indexed by *slot number*. 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 C++ class Classifier provides a base class from which other classifiers are derived, the classifier is defined as follows.

```cpp
class Classifier : public NsObject {
public:
    Classifier();
    void recv(Packet*, Handler* h = 0);
protected:
    Classifier();
    void install(int slot, NsObject*);
    void clear(int slot);
    virtual int command(int argc, const char*const* argv);
    virtual int classify(Packet*const) = 0;
    void alloc(int);
    NsObject** slot_; /* table that maps slot number to a NsObject */
    int nslot_;    /* table that maps slot number to a NsObject */
    int maxslot_;}
```

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

```cpp
/*
 * 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;
}
```
int cl = classify(p);
if (cl < 0 || cl >= nslot_ || (node = slot_[cl]) == 0) {
  Tcl::instance().evalf("%s no-slot %d", name(), cl);
  Packet::free(p);
  return;
}
node->recv(p);

int Classifier::command(int argc, const char*const* argv)
{
  Tcl& tcl = Tcl::instance();
  if (argc == 3) {
    /*
     * $classifier clear $slot
     */
    if (strcmp(argv[1], "clear") == 0) {
      int slot = atoi(argv[2]);
      clear(slot);
      return (TCLOK);
    }
    /*
     * $classifier installNext $node
     */
    if (strcmp(argv[1], "installNext") == 0) {
      int slot = maxslot_ + 1;
      NsObject* node = (NsObject*)TclObject::lookup(argv[2]);
      install(slot, node);
      tcl.resultf("%u", slot);
      return TCL_OK;
    }
    if (strcmp(argv[1], "slot") == 0) {
      int slot = atoi(argv[2]);
      if ((slot >= 0) && (slot < nslot_)) {
        tcl.resultf("%s", slot_[slot]->name());
        return TCL_OK;
      }
  }
When a classifier recv()'s a packet, it hands it to the classify() method. The classify() method will determine and return a slot index into the table of slots. If the index is valid, and points to a valid TclObject, the classifier will hand the packet to that object using object's recv() method. If the index is not valid, the classifier will invoke the instance procedure no-slot{} to attempt to populate the table correctly.

Address Classifiers

An address classifier is used in supporting packet forwarding. It applies a bitwise 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) {
        bind("mask_", (int*)&mask_);
        bind("shift_", &shift_);
    }

protected:
    int classify(Packet *const p) {

163
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. The mask_ and shift_ values are set through OTcl.

Multicast Classifiers

The multicast classifier classifies packets according to both source and destination (group) addresses. It maintains a (chained hash) table mapping source/group pairs to slot numbers. When a packet arrives containing a source/group unknown to the classifier, it invokes an Otcl procedure Node::new-group() to add an entry to its table. This OTcl procedure may use the method set-hash to add new (source, group, slot) 3-tuples to the classifier’s table. The multicast classifier is defined in ~ns/classifier-mcast.cc as follows:

```cpp
static class MCastClassifierClass : public TclClass {
    public:
        MCastClassifierClass() : TclClass("Classifier/Multicast") {}
        TclObject* create(int argc, const char*const* argv) {
            return (new MCastClassifier());
        }
    } class_mcast_classifier;

class MCastClassifier : public Classifier {
    public:
        MCastClassifier();
        ~MCastClassifier();
    protected:
        int command(int argc, const char*const* argv);
        int classify(Packet *const p);
        int findslot();
        void set_hash(nsaddr_t src, nsaddr_t dst, int slot);
        int hash(nsaddr_t src, nsaddr_t dst) const {
```
The class MCastClassifier implements a chained hash table and applies a hash function on both the packet source and destination addresses. The hash function returns the slot
number to index the slot_ table in the underlying object. A hash miss implies packet
delivery to a previously-unknown group; OTcl is called to handle the situation. The OTcl
code is expected to insert an appropriate entry into the hash table.

**MultiPath Classifier**

This object is devised to support equal cost multipath forwarding, where the node has
multiple equal cost routes to the same destination, and would like to use all of them
simultaneously. This object does not look at any field in the packet. With every
succeeding packet, it simply returns the next filled slot in round robin fashion. The
definitions for this classifier are shown below:

```cpp
class MultiPathForwarder : public Classifier {
public:
    MultiPathForwarder() : ns_(0), Classifier() {} 
    virtual int classify(Packet* const) {
        int cl;
        int fail = ns_; 
        do {
            cl = ns_++;
            ns_ %= (maxslot_ + 1);
        } while (slot_[cl] == 0 && ns_ != fail);
        return cl;
    }
private:
    int ns_; /* next slot to be used. Probably a misnomer? */
};
```

**Hash Classifier**

This object is used to classify a packet as a member of a particular flow. As their name
indicates, hash classifiers use a hash table internally to assign packets to flows. These
objects are used where flow-level information is required (e.g. in flow-specific queuing
disciplines and statistics collection). Several "flow granularities" are available. In
particular, packets may be assigned to flows based on flow ID, destination address,
source/destination addresses, or the combination of source/destination addresses plus flow ID. The fields accessed by the hash classifier are limited to the ip header: src(), dst(), flowid().

The following constructors are used for the hash classifiers:

- Classifier/Hash/SrcDest
- Classifier/Hash/Dest
- Classifier/Hash/Fid
- Classifier/Hash/SrcDestFid

The hash classifier receives packets, classifies them according to their flow criteria, and retrieves the classifier slot indicating the next node that should receive the packet. The methods for a hash classifier are as follows:

- \$\text{hashcl set-hash buck src dst fid slot}
- \$\text{hashcl lookup buck src dst fid}
- \$\text{hashcl del-hash src dst fid}
- \$\text{hashcl resize nbuck}

**Node Interface**

Following is a list of common node commands used in simulation scripts:

- \$\text{ns_ node [<hier_addr>] /* to create and return a node instance. If <hier_addr> is given, assign the node address to be <hier_addr> */}
- \$\text{ns_ node-config -<config-parameter> <optional-val> /* configure nodes */.}
- \$\text{node id /* Returns the id number of the node */}
- \$\text{node node-addr /* Returns the address of the node */}
- \$\text{node reset /* Resets all agent attached to this node */}
- \$\text{node agent <port_num> /* Returns the handle of the agent at the specified port. If no agent is found at the given port, a null string is returned */}
- \$\text{node entry /* Returns the entry point for the node. This is first object that handles packet receiving at this node */}
- \$\text{node attach <agent> <optional:port_num> /* Attaches the <agent> to this node*/}
- \$\text{node neighbors /* This returns the list of neighbors for the node */}
$node add-neighbor <neighbor_node> /* This is a command to add <neighbor_node> to the list of neighbors maintained by the node */

Following is a list of internal node methods:
$node add-route <destination_id> <target> /* This is used in routing to populate the classifier. The target is a head of the link for that destination */
$node alloc-port <null_agent> /* This returns the next available port number */
$node incr-rtgtable-size /* This command is used to increase the routing-table size every time a routing-entry is added to the classifiers */

**Link Procedures** The Otcl class SimpleLink implements a simple point-to-point link with an associated queue and delay. It is derived from the base Otcl class Link as follows:

```otcl
Class SimpleLink -superclass Link
SimpleLink instproc init { src dst bw delay q { Utype "DelayLink" } } {
    $self next $src $dst
    $self instvar link_ queue_ head_ toNode_ ttl_

    ... queue_ $q
    set link_ [new Delay/Link]
    $link_ set bandwidth_ $bw
    $link_ set delay_ $delay
    $queue_ target $link_
    $link_ target [$toNode_ entry]

    ... # XXX
    # put the ttl checker after the delay
    # so we don’t have to worry about accounting
    # for ttl-drops within the trace and/or monitor
    # fabric
    #
    set ttl_ [new TTLChecker]
    $ttl_ target [$link_ target]
    $link_ target $ttl_
}
```

When a SimpleLink object is created, new Delay/Link and TTLChecker objects are also created.
APPENDIX-B

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.

The C++ Queue Class

The Queue class is derived from a Connector base class. It provides a base class used by particular types of (derived) queue classes, as well as a call-back function to implement blocking. The following definitions are provided in queue.h:

class Queue : public Connector {
    public:
        virtual void enque(Packet*) = 0;
        virtual Packet* deque() = 0;
        void recv(Packet*, Handler*);
        void resume();
        int blocked();
        void unblock();
        void block();
    protected:
        Queue();
        int command(int argc, const char*const* argv);
        int qlim_; /* maximum allowed pkts in queue */
        int blocked_;
        int unblock_on_resume_; /* unblock q on idle? */
        QueueHandler qh_;
};
The enqueue and dequeue 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. Particular queues do not, in general, override the recv function because it invokes the the particular enqueue and dequeue.

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 (most of the time if the queue is occupied). A blocked queue will remain blocked as long as it 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 means of a downstream neighbor scheduling it via a callback), when no packets are queued. The callback is implemented by using the following class and methods:

```cpp
class QueueHandler : public Handler {
   public:
      inline QueueHandler(Queue& q) : queue_(q) {}
      void handle(Event*) { /* calls queue_.resume() */
   private:
      Queue& queue_
   }
   void QueueHandler::handle(Event*)
   {
      queue_.resume();
   }
   Queue::Queue() : drop_(0), blocked_(0), qh_(*this)
   {
      Tcl& tcl = Tcl::instance();
      bind("limit_", &qlim_);
   }
   void Queue::recv(Packet* p, Handler*)
   {
      enqueue(p);
   }
};
```
if (!blocked_) {
    /*
     * We're not block. Get a packet and send it on.
     * We perform an extra check because the queue
     * might drop the packet even if it was
     * previously empty! (e.g., RED can do this.)
     */
    p = dequeue();
    if (p != 0) {
        blocked_ = 1;
        target_->recv(p, &qh_);
    }
}

void Queue::resume()
{
    // Get a packet and send it on.
    Packet* p = dequeue();
    if (p != 0)
        target_->recv(p, &qh_);
    else {
        if (unblock_on_resume_)
            blocked_ = 0;
        else
            blocked_ = 1;
    }
}

When a new Queue object is created, it includes a QueueHandler object (qh_) which is initialized to contain a reference to the new Queue object (Queue& QueueHandler::queue_). This is performed by the Queue constructor using the expression qh_(*this). When a Queue receives a packet it calls the subclass (i.e. queueing discipline-specific) version of the enqueue function with the packet. If the queue is not blocked, it is allowed to send a packet and calls the specific dequeue function which determines which packet to send, blocks the queue (because a packet is now in transit), and sends the packet to the queue's downstream neighbor.
PacketQueue Class

The Queue classes implement buffer management and scheduling and the PacketQueue class is used for this purpose, and is defined as follows:

```cpp
class PacketQueue {
public:
    PacketQueue();
    int length(); /* queue length in packets */
    void enqueue(Packet* p);
    Packet* dequeue();
    Packet* lookup(int n);
    /* remove a specific packet, which must be in the queue */
    void remove(Packet*);
protected:
    Packet* head_;  // packet count
    Packet** tail_;  // packet count
};
```

This class maintains a linked-list of packets, and is commonly used by particular scheduling and buffer management disciplines to hold an ordered set of packets. The PacketQueue class maintains current counts of the number of packets held in the queue which is returned by the length() method. The enqueue function places the specified packet at the end of the queue and updates the len_ member variable.

The dequeue function returns the packet at the head of the queue and removes it from the queue (and updates the counters), or returns NULL if the queue is empty. The lookup function returns the nth packet from the head of the queue, or NULL otherwise. The remove function deletes the packet stored in the given address from the queue (and updates the counters). It causes an abnormal program termination if the packet does not exist. The following program illustrates the implementation of the Queue/DropTail object, and declare the class and its OTcl linkage:
class DropTail: public Queue {
protected:
    Packet* deque();
    PacketQueue q_; 
};

The base class Queue, from which DropTail is derived, provides most of the needed functionality. The drop-tail queue maintains exactly one FIFO queue, implemented by including an object of the PacketQueue class. Drop-tail implements its own versions of enque and deque as follows:

```c++
/*
 * drop-tail
 */
void DropTail::enqueue(Packet* p)
{
    q_.enqueue(p);
    if (q_.length() >= qlim_) {
        q_.remove(p);
        drop(p);
    }
}
Packet* DropTail::deque()
{
    return (q_.dequeue());
}
```

Here, the enqueue 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.
Queue Monitoring

Queue monitoring refers to the capability of tracking the dynamics of packets at a queue (or other object). A queue monitor tracks packet arrival/departure/drop statistics, and optionally compute averages of these values.

A QueueMonitor is defined as follows (~ns/queue-monitor.cc):

class QueueMonitor : public TclObject {
   public:
      QueueMonitor() : bytesInt_(NULL), pktsInt_(NULL),
      delaySamp_(NULL),
      size_(0), pkts_(0),
      parrivals_(0), barrivals_(0),
      pdepartures_(0), bdepartures_(0),
      pdrops_(0), bdrops_(0),
      srcId_(0), dstId_(0), channel_(0) {
         bind("size_", &size_);
         bind("pkts_", &pkts_);
         bind("parrivals_", &parrivals_);
         bind("barrivals_", &barrivals_);
         bind("pdepartures_", &pdepartures_);
         bind("bdepartures_", &bdepartures_);
         bind("pdrops_", &pdrops_);
         bind("bdrops_", &bdrops_);
         bind("off_cmn_", &off_cmn_);
      }

      int size() const { return (size_); }
      int pkts() const { return (pkts_); }
      int parrivals() const { return (parrivals_); }
      int barrivals() const { return (barrivals_); }
      int pdepartures() const { return (pdepartures_); }
      int bdepartures() const { return (bdepartures_); }
      int pdrops() const { return (pdrops_); }
      int.bdrops() const { return (bdrops_); }

174
void printStats();
virtual void in(Packet*);
virtual void out(Packet*);
virtual void drop(Packet*);
virtual void edrop(Packet*) { abort(); }; // not here
virtual int command(int argc, const char* const* argv);
...

// packet arrival to a queue
void QueueMonitor::in(Packet* p)
{
    hdr_cmn* hdr = (hdr_cmn*)p->access(off_cmn_);
double now = Scheduler::instance().clock();
int pktsz = hdr->size();
barrivals_ += pktsz;
parrivals_++;
size_ += pktsz;
pkts_++;
if (bytesInt_)
    bytesInt_->newPoint(now, double(size_));
if (pktsInt_)
    pktsInt_->newPoint(now, double(pkts_));
if (delaySamp_)
    hdr->timestamp() = now;
if (channel_)
    printStats();
}

All bound variables beginning with p refer to packet counts, and all variables beginning with b refer to byte counts. The variable size_ records the instantaneous queue size in bytes, and the variable pkts_ records the same value in packets. When a QueueMonitor is configured to include the integral functions (on bytes or packets or both), it computes the approximate integral of the queue size (in bytes) with respect to time over the interval \([t_0, now]\), where \(t_0\) is either the start of the simulation or the last time the sum_ field of the underlying Integrator class was reset.
Per-Flow Monitoring

A collection of specialized classes are used to implement per-flow statistics gathering. These classes include: QueueMonitor/ED/Flowmon, QueueMonitor/ED/Flow, and Classifier/Hash. Typically, an arriving packet is inspected to determine to which flow it belongs. Once the correct flow is determined, the packet is passed to a flow monitor, which is responsible for collecting per-flow state. Per-flow state is contained in flow objects in a one-to-one relationship to the flows known by the flow monitor. It provides the following OTcl interface:

- classifier get(set) classifier to map packets to flows
- attach attach a Tcl I/O channel to this monitor
- dump dump contents of flow monitor to Tcl channel
- flows return string of flow object names known to this monitor

The flow monitor defines a trace format which may be used by post-processing scripts to determine various counts on a per-flow basis. The format is defined by the following code in ~/ns/flowmon.cc:

```c
void
FlowMon::fFormat(Flow* f)
{
    double now = Scheduler::instance().clock();
    sprintf(wrk_, "%8.3f%7d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d %6d`
f->ebdrops(), // early drops this flow (bytes)
parivals(), // all arrivals (pkts)
barrivals(), // all arrivals (bytes)
epdrops(), // total early drops (pkts)
ebdrops(), // total early drops (bytes)
pdrops(), // total drops (pkts)
bdrops(), // total drops (bytes)
f->pdrops(), // drops this flow (pkts) [includes edrops]
f->bdrops() // drops this flow (bytes) [includes edrops]
};

Its OTcl interface contains only bound variables:

src_ source address on packets for this flow
dst_ destination address on packets for this flow
flowid_ flow id on packets for this flow

The class QueueMonitor/Flow is used by the flow monitor for containing per-flow counters. As a subclass of QueueMonitor, it inherits the standard counters for arrivals, departures, and drops, both in packets and bytes.