The NS (v2) Simulator Workshop

brought to you by

Kevin Fall
Lawrence Berkeley National Laboratory
kfall@ee.lbl.gov
http://www-nrg.ee.lbl.gov/kfall

AND

Kannan Varadhan
USC/ISI
kannan@catarina.usc.edu
http://www.isi.edu/~kannan

September 18, 1997

Audience and Outline

- Audience
  - network researchers
  - educators
  - developers
- Topics for today
  - VINT project goals and status (Sally)
  - architecture plus some history (Steven)
  - overview of major components (Kevin)
  - project/code status (Kevin)
  - details of major components (Kevin)
  - C++/OTcl linkage and simulation debugging (Kannan)
  - topology generation and session-level support (Kannan)
  - multicast and reliable multicast (Kannan)
  - a complex link: CBQ (Kevin)
  - discussion and futures (Everyone)

NSv2 Architecture

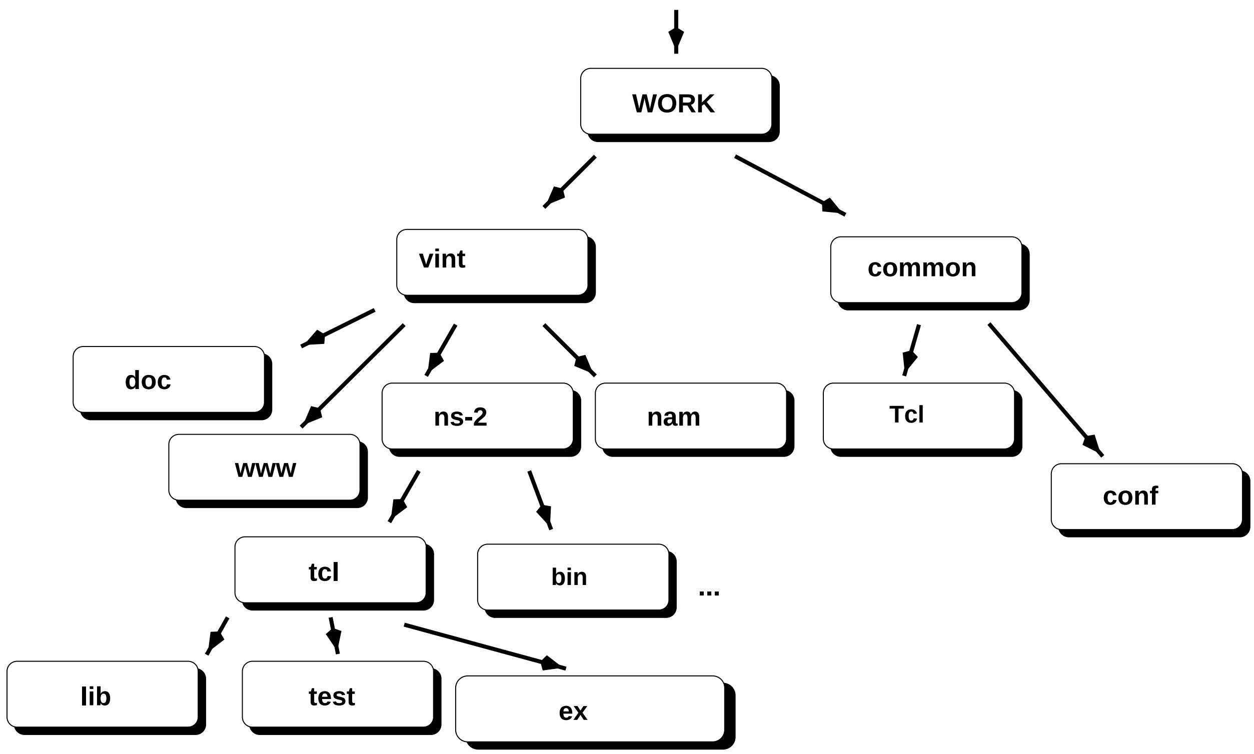
- Object-oriented structure
  - evolution from NSv1 (C++ with regular Tcl)
  - objects implemented in C++ and "OTcl"
  - OTcl: object-oriented extension to Tcl
    (from David Wetherall at MIT/LCS for VuSystem)
- Control/"Data" separation
  - control operations in OTcl
  - data pass through C++ objects (for speed)
- Modular approach
  - fine-grain object decomposition
  - positives: composible, re-usable
  - negatives: must "plumb" in OTcl,
    developer must be comfortable with both environments, tools

Development Status

- simulator code basis for VINT Project
- 5ish people actively contributing to the code base
- other contributions from Xerox PARC, USC/ISI, UCB, LBNL
- Some approximate numbers:
  - 27K lines of C++ code
  - 12K lines of OTcl support code
  - 18K lines of test suites, examples
  - 5K lines of documentation!
- See main VINT and NS-2 web pages at:
  `http://netweb.usc.edu/vint`
  `http://www-mash.cs.berkeley.edu/ns/ns.html`
- Open mailing lists:
  - ns-users@mash.cs.berkeley.edu
  - ns-announce@mash.cs.berkeley.edu
- To subscribe:
  - majordomo@mash.cs.berkeley.edu

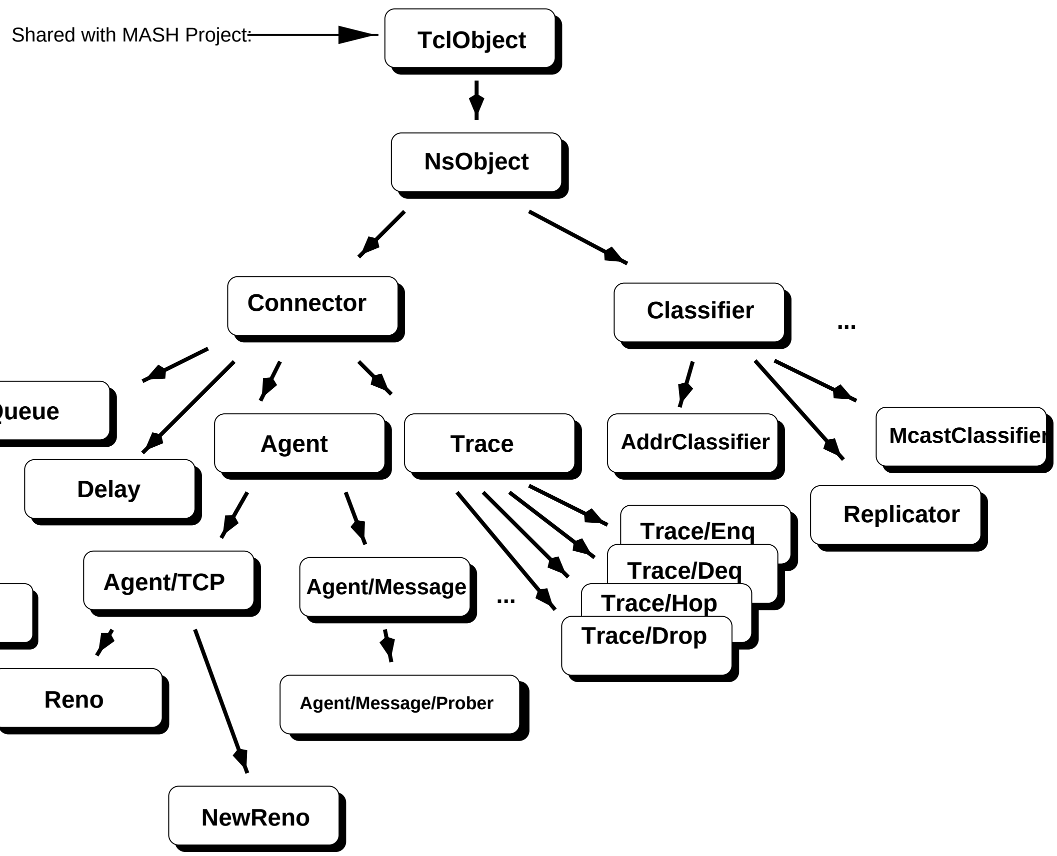
Directory Structure

- `common` directory shared between MASH (UCB) and VINT projects



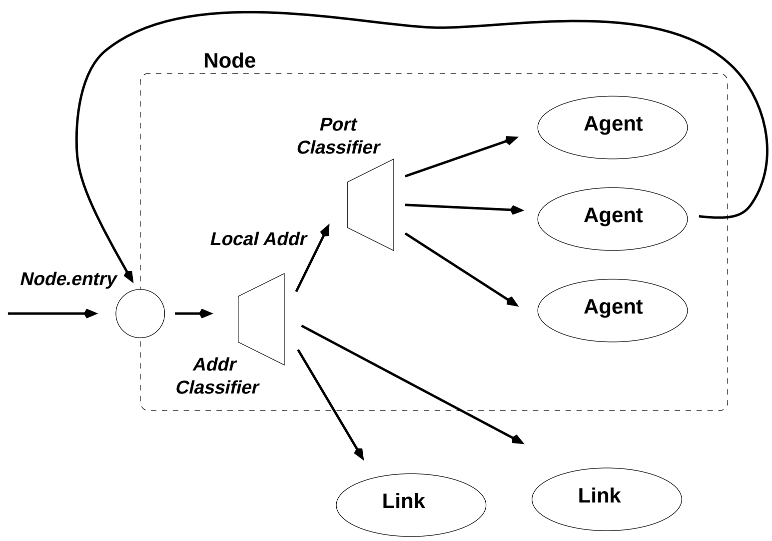
Class Hierarchy

- Top-level classes implement simple abstractions:



Example: a node

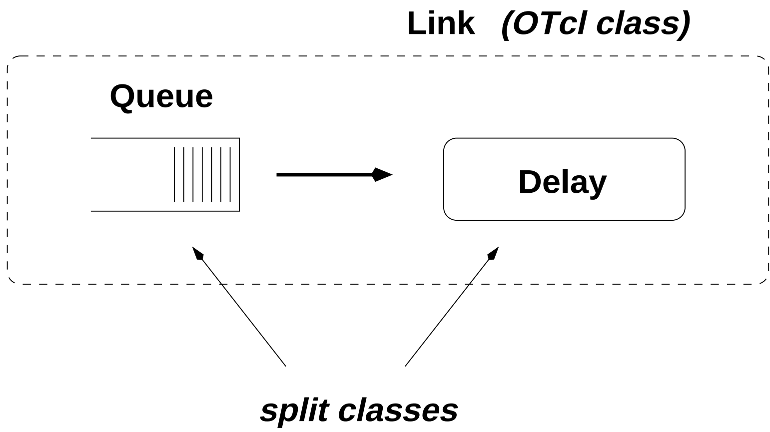
- Node: a collection of *agents* and *classifiers*
- Agents: usually protocol endpoints and related objects
- Classifiers: packet demultiplexers



- Note that the node “routes” to itself or to downstream links

Example: a link

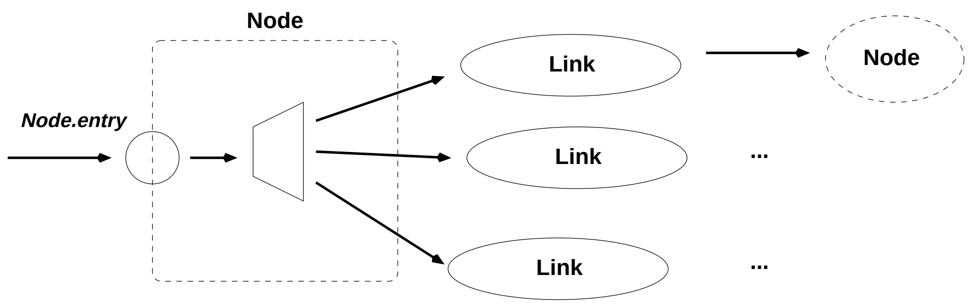
- keeps track of "from" and "to" Node objects
- generally encapsulates a queue, delay and possibly ttl checker



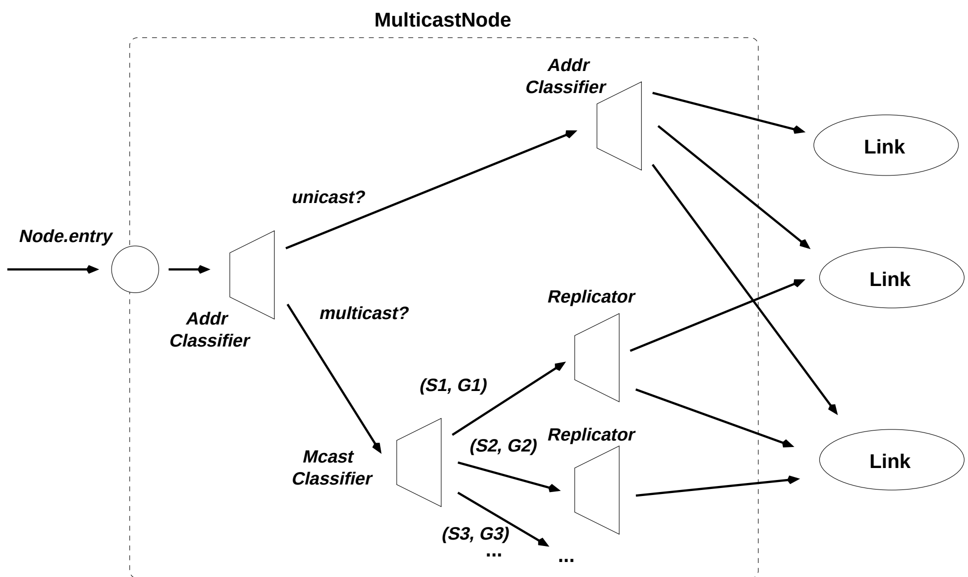
- Many more complex objects built from this base

Example: routers

- routers (unicast and multicast) by "plumbing"



- multicast router adds additional classifiers and replicators
- Replicators: demuxers with multiple fanout



OTcl Basics

- See the page at `ftp://ftp.tns.lcs.mit.edu/pub/otcl/`
- object oriented extension to tcl
- classes are objects with support for inheritance
- Analogs to C++:
  - C++ has single class decl ⇒
    OTcl attaches methods to object or class
  - C++ constructor/destructor ⇒ OTcl init/destroy methods
  - *this* ⇒ $*self*
  - OTcl methods always "virtual"
  - C++ shadowed methods called explicitly with scope operator
    ⇒ OTcl methods combined implicitly with $*self next*
  - C++ static variables ⇒ OTcl class variables
  - (multiple inheritance is supported)

OTcl Basics (contd)

- use *instvar* and *instproc* to define/access member functions and variables
- Example:

```
Class Counter
Counter instproc init {} {
        $self instvar cnt_
        set cnt_ 0
}
Counter instproc bump {} {
        $self instvar cnt_
        incr cnt_
}
Counter instproc val {} {
        $self instvar cnt_
        return $cnt_
}

Counter c
c val → 0
c bump
c val → 1
```

C++/OTcl Split Objects

- Split objects: implement methods in either language
- *new* and *delete*

```
set c [new Counter]
$c val -> 0
$c bump
$c val -> 1
delete $c
```

- Define instance variables in either C++ or OTcl:

```
Counter::Counter()
{
        bind("cnt_",
&value_);
        value_ = 10;
        ...
}
```

vs.

```
$self set cnt_ 10
```

bind() simply uses *Tcl_TraceVar*

Example: a simple simulation

- A small but complete simulation script:
  - set up 4-node topology and one bulk-data transfer TCP
  - arrange to trace the queue on the r1-k1 link
  - place trace output in the file `simp.out.tr`
-
```
# Create a simple four node topology:
#          s1
#            \
#      8Mb,5ms \   0.8Mb,50ms
#               r1 --------- k1
#      8Mb,5ms /
#            /
#          s2
set stoptime 10.0
set ns [new Simulator]
set node_(s1) [$ns node]
set node_(s2) [$ns node]
set node_(r1) [$ns node]
set node_(k1) [$ns node]
$ns duplex-link $node_(s1) $node_(r1) 8Mb 5ms DropTail
$ns duplex-link $node_(s2) $node_(r1) 8Mb 5ms DropTail
$ns duplex-link $node_(r1) $node_(k1) 800Kb 50ms DropTail
$ns queue-limit $node_(r1) $node_(k1) 6
$ns queue-limit $node_(k1) $node_(r1) 6
set tcp1 [$ns create-connection TCP $node_(s1) TCPSink $node_(k1) 0]
$tcp1 set window_ 50
$tcp1 set packetSize_ 1500

# Set up FTP source
set ftp1 [$tcp1 attach-source FTP]
set tf [open simp.out.tr w]
$ns trace-queue $node_(r1) $node_(k1) $tf
$ns at 0.0 "$ftp1 start"
$ns at $stoptime "close $tf; puts \"simulation complete\";  $ns halt"
$ns run
```

Example: a simple simulation (cont)

- The trace file produced looks like this:

```
+ 0.0065 2 3 tcp 1500 ------ 0 0.0 3.0 0 0
- 0.0065 2 3 tcp 1500 ------ 0 0.0 3.0 0 0
+ 0.23344 2 3 tcp 1500 ------ 0 0.0 3.0 1 2
- 0.23344 2 3 tcp 1500 ------ 0 0.0 3.0 1 2
+ 0.23494 2 3 tcp 1500 ------ 0 0.0 3.0 2 3
- 0.24844 2 3 tcp 1500 ------ 0 0.0 3.0 2 3
+ 0.46038 2 3 tcp 1500 ------ 0 0.0 3.0 3 6
- 0.46038 2 3 tcp 1500 ------ 0 0.0 3.0 3 6
+ 0.46188 2 3 tcp 1500 ------ 0 0.0 3.0 4 7
+ 0.47538 2 3 tcp 1500 ------ 0 0.0 3.0 5 8
...
+ 0.98926 2 3 tcp 1500 ------ 0 0.0 3.0 25 40
+ 0.99076 2 3 tcp 1500 ------ 0 0.0 3.0 26 41
d 0.99076 2 3 tcp 1500 ------ 0 0.0 3.0 26 41
- 1.00426 2 3 tcp 1500 ------ 0 0.0 3.0 21 36
+ 1.00426 2 3 tcp 1500 ------ 0 0.0 3.0 27 42
+ 1.00576 2 3 tcp 1500 ------ 0 0.0 3.0 28 43
d 1.00576 2 3 tcp 1500 ------ 0 0.0 3.0 28 43
- 1.01926 2 3 tcp 1500 ------ 0 0.0 3.0 22 37
+ 1.01926 2 3 tcp 1500 ------ 0 0.0 3.0 29 44
+ 1.02076 2 3 tcp 1500 ------ 0 0.0 3.0 30 45
d 1.02076 2 3 tcp 1500 ------ 0 0.0 3.0 30 45
- 1.03426 2 3 tcp 1500 ------ 0 0.0 3.0 23 38
- 1.04926 2 3 tcp 1500 ------ 0 0.0 3.0 24 39
- 1.06426 2 3 tcp 1500 ------ 0 0.0 3.0 25 40
...
```

The Simulator

- Simulator API is a set of methods belonging to a *simulator* object:
- Create a simulator with:

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

- What this does:
  - initialize the packet format (calls `create_packetformat`)
  - create a scheduler (defaults to a simple linked-list scheduler)
- Scheduler:
  - handles time, timers and events (packets), deferred executions ("ATs")
  - `Scheduler/List` - linked-list scheduler
  - `Scheduler/Heap` - heap-based scheduler
  - `Scheduler/Calendar` - calendar-queue scheduler
  - see Reeves, "Complexity Analyses of Event Set Algorithms", *The Computer Journal*, 27(1), 1984

Using the scheduler

- Scheduler API is through Simulator object:

  `Simulator instproc now` ;# *return scheduler's notion of current time*
  `Simulator instproc at args` ;# *schedule execution of code at specified time*
  `Simulator instproc run args` ;# *start scheduler*
  `Simulator instproc halt` ;# *stop (pause) the scheduler*
  `Simulator instproc create-trace type files src dst` ;# *create trace object*
  `Simulator instproc create_packetformat` ;# *set up the simulator's packet format*

- Example:

```
MySim instproc begin {} {
        ...
        set ns_ [new Simulator]
        $ns_ use-scheduler Heap
        $ns_ at 300.5 "$self complete_sim"
        ...
}
MySim instproc complete_sim {} {
        ...
}
```

Simulator Timing

- each object has a generic receive method
  `NsObject::recv(Packet*, Handler* h = 0)`
- most objects have single neighbor `Connector::target_`
- *cut-through* transfers; send packet directly to neighbor without involving scheduler
  `Connector::send(Packet* p) { target_->recv(p); }`
- *barrier*:
  - any point that advances time into future
    (i.e., delay element)
  - need inter-object "protocol" to decouple timing
  - barrier takes non-null Handler
  - schedule delay and invoke handler on completion
  - example: queue/delay objects (later)

Packets

- packets are *events* (may be scheduled to "arrive")
- contain header section and (sometimes) data
- header section is a cascade of all in-use headers
- all packets contain a *common header*:
  - packet size - used to compute transmission time
  - timestamp, type, uid, interface label
    (for debugging, and multicast routing)
- new protocol agents may need to define new headers

Packet Header Format

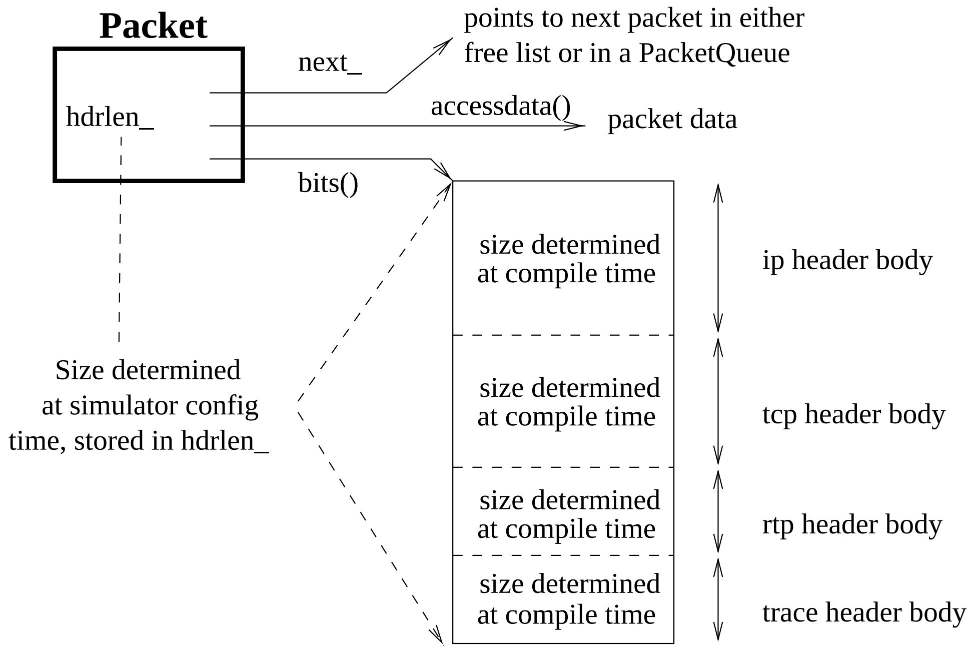


Figure 1: A Packet Object

- header contents are constructed at simulator initialization time
- performed by `create_packetformat`

Connectors

- *Connector:* simple in/out topology object with "drop target"

```
/*
 * An NsObject with only a single neighbor.
 */
class Connector : public NsObject {
public:
        Connector();
        inline NsObject* target() { return target_; }
        virtual void drop(Packet* p);
protected:
        int command(int argc, const char*const* argv);
        void recv(Packet*, Handler* callback = 0);
        inline void send(Packet* p, Handler* h) { target_->recv(p, h); }

        NsObject* target_;
        NsObject* drop_;        // drop target for this connector
};
```

- if drop target undefined, dropped packets are freed

Error Models

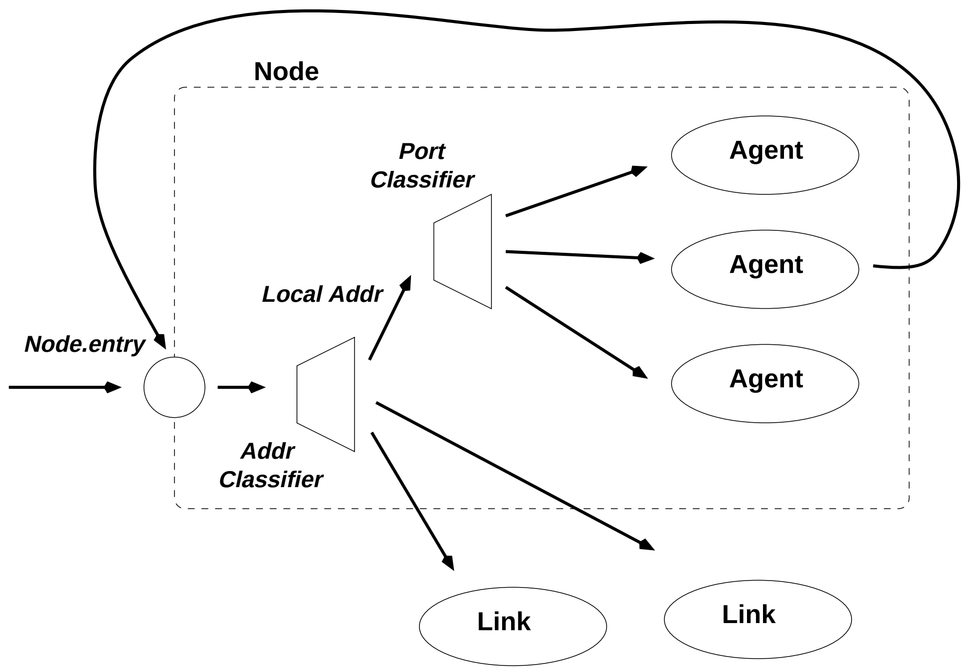
- *Error Model:* a (simple) parameterized lossy connector (can be used as a base class for other loss models)
- drops packet or sets “error” bit (in common header)
- error *units*: packets, bits, time

```
Usage:
        # create a loss_module and set its packet error rate to 1 percent
        set loss_module [new ErrorModel]
        $loss_module set rate_ 0.01

        # optional:  set the unit and random variable
        $em unit pkt            # error unit: packets (the default)
        $em ranvar [new RandomVariable/Uniform]

        # set target for dropped packets
        $loss_module drop-target [$ns_ set nullAgent_]
```

- if drop target undefined, dropped packets are freed

Agents

- *Agents:* usually a protocol endpoint/entity (but may also be used for implementing routing protocols)
- Where they fit in:



- What they provide:
  - a local and destination address (like an IP-layer sender)
  - functions for helping to generate/fill-in in packet fields

Creating a new Agent

- The `Agent` class:

```
class Agent : public Connector {
 public:
        Agent(int pktType);
        virtual ~Agent();
        virtual void timeout(int tno);
 protected:
        int command(int argc, const char*const* argv);
        void recv(Packet*, Handler*);
        ...
```

- basic tasks to create a new agent:

  1. decide its inheritance structure
  2. create the class, **`recv`**, and **`timeout`** functions (if needed)
  3. define OTcl linkage functions (Kannan will explain how later)
  4. write the necessary OTcl code to access your agent

- hardest part may be understanding the OTcl/C++ interaction (fortunately, much of this is shielded from you if you so choose)

Example: the Message Agent

- provides a very simple place to store a message
- Packet header (from `message.h`):

```
struct hdr_msg {
        char msg_[64];
        /* per-field member functions */
        char* msg() { return (msg_); }
        int maxmsg() { return (sizeof(msg_)); }
};
```

- OTcl linkage (for class creation, from `message.cc`):

```
static class MessageHeaderClass : public PacketHeaderClass {
public:
        MessageHeaderClass() :
             PacketHeaderClass("PacketHeader/Message",
             sizeof(hdr_msg)) {}
} class_msghdr;
```

Example: the Message Agent (cont)

- The class definition, constructor and variable linkage:

```
static class MessageClass : public TclClass {
public:
        MessageClass() : TclClass("Agent/Message") {}
        TclObject* create(int, const char*const*) {
                return (new MessageAgent());
        }
} class_message;

class MessageAgent : public Agent {
 public:
        MessageAgent();
        int command(int argc, const char*const* argv);
        void recv(Packet*, Handler*);
protected:
        int off_msg_;
};
MessageAgent::MessageAgent() : Agent(PT_MESSAGE)
{
        bind("packetSize_", &size_);
        bind("off_msg_", &off_msg_);
}
```

Example: the Message Agent (cont)

- Main functions:

```
void MessageAgent::recv(Packet* pkt, Handler*)
{
        hdr_msg* mh = (hdr_msg*)pkt->access(off_msg_);
        ... process packet ...
}

int MessageAgent::command(int argc, const char*const* argv)
{
        Tcl& tcl = Tcl::instance(); // call into interp
        if (argc == 3) { // $obj send msgtext
                if (strcmp(argv[1], "send") == 0) {
                        Packet* pkt = allocpkt();
                        hdr_msg* mh = (hdr_msg*)pkt->access(off_msg_);
                        const char* s = argv[2];
                        int n = strlen(s);
                        if (n >= mh->maxmsg()) {
                                tcl.result("message too big");
                                Packet::free(pkt);
                                return (TCL_ERROR);
                        }
                        strcpy(mh->msg(), s);
                        send(pkt, 0);
                        return (TCL_OK);
                }
        }
        return (Agent::command(argc, argv)); // for inheritance
}
```

TCP Agents

- ns has several variants of TCP available:
  - Agent/TCP - a "tahoe" TCP sender
  - Agent/TCP/Reno - a "Reno" TCP sender
  - Agent/TCP/NewReno - Reno with a modification
  - Agent/TCP/Sack1 - TCP with selective repeat (follows RFC2018)
  - Agent/TCP/Vegas - TCP Vegas
  - Agent/TCP/Fack - Reno TCP with "forward acknowledgement"
- The one-way TCP receiving agents currently supported are:
  - Agent/TCPSink - TCP sink with one ACK per packet
  - Agent/TCPSink/DelAck - TCP sink with configurable delay per ACK
  - Agent/TCPSink/Sack1 - selective ACK sink (follows RFC2018)
  - Agent/TCPSink/Sack1/DelAck - Sack1 with DelAck
- The two-way experimental sender currently supports only a Reno form of TCP:
  - Agent/TCP/FullTcp

Base TCP Agents

- `TCP` (Tahoe), `TCP/Reno`, and `TCP/NewReno`
- Common features:
  - computations all in packet units w/configurable packet size
  - fast retransmit
  - slow-start and congestion avoidance
  - dynamic RTT estimation and RTX timeout assignment
  - simulated (constant) receiver's advertised window
- Tahoe TCP:
  - perform slow-start on any loss (RTO or fast retransmit)
  - no fast recovery
- Reno TCP:
  - fast *recovery*: inflate *cwnd* by dup ack count until new ACK
  - slow-start on RTO
  - on fast retransmit:

    $$cwnd \leftarrow curwin/2 \;,\; ssthresh \leftarrow cwnd$$
- "Newreno" TCP:
  - modest modification to Reno TCP
  - only exit fast recovery after ACK for highest segment arrives
  - helps reduce "stalling" due to multiple packet drops in a window

Other TCP Agents

- `TCP/Sack`, `TCP/Fack`, and `TCP/Vegas`
- Selective ACK TCP:
  - SACK simulation based on RFC2018
  - ACKs carry extra information indicating received segments
  - requires SACK-aware sink
  - sender avoids sending redundant info
  - default to 3 "SACK blocks" (for using timestamps, see RFC2018)
    - block contains start/end sequence numbers
    - block containing most recently received segment always present
  - regular ACK number still gives final say
- Fack TCP:
  - "forward ACK" TCP (experimental, see SIGCOMM '96)
  - use SACK info for estimate of packets in the network
  - overdamping algorithm (to limit slow-start overshoot)
  - rampdown algorithm (for transmission smoothing)
- Vegas TCP:
  - contributed code from Ted Kuo (NC State Univ)
  - not directly supported at this time

TCP Agent Parameters

- Common configuration parameters and defaults for TCP agents:

```
Agent/TCP set window_   20                  ;# max bound on window size
Agent/TCP set windowInit_ 1                 ;# initial/reset value of cwnd
Agent/TCP set windowOption_ 1               ;# cong avoid algorithm (1: standard)
Agent/TCP set windowConstant_ 4             ;# used only when windowOption != 1
Agent/TCP set windowThresh_ 0.002           ;# used in computing averaged window
Agent/TCP set overhead_ 0                   ;# !=0 adds random time between sends
Agent/TCP set ecn_ 0                        ;# TCP should react to ecn bit
Agent/TCP set packetSize_ 1000              ;# packet size used by sender (bytes)
Agent/TCP set bugFix_ true                  ;# see documentation
Agent/TCP set slow_start_restart_ true      ;# see documentation
Agent/TCP set tcpTick_ 0.1                  ;# timer granularity in sec (.1 is NONST
ANDARD)
Agent/TCP set maxrto_ 64                    ;# bound on RTO (seconds)
Agent/TCP set dupacks_ 0                    ;# duplicate ACK counter
Agent/TCP set ack_ 0                        ;# highest ACK received
Agent/TCP set cwnd_ 0                       ;# congestion window (packets)
Agent/TCP set awnd_ 0                       ;# averaged cwnd (experimental)
Agent/TCP set ssthresh_ 0                   ;# slow-stat threshold (packets)
Agent/TCP set rtt_ 0                        ;# rtt sample
Agent/TCP set srtt_ 0                       ;# smoothed (averaged) rtt
Agent/TCP set rttvar_ 0                     ;# mean deviation of rtt samples
Agent/TCP set backoff_ 0                    ;# current RTO backoff factor
Agent/TCP set maxseq_ 0                     ;# max (packet) seq number sent
```

TCP Sink Agents

- Sinks for one-way TCP senders
- Types
  - standard sinks, delayed-ACK sinks, SACK sinks
- Standard sinks:
  - generate one ACK per packet received
  - ACK number overloaded in "sequence number" packet field
- Delayed-ACK sinks:
  - same as standard, but with variable delay added between ACKs
  - time to delay ACKs specified in seconds
- SACK sinks:
  - generates additional information for SACK capable sender
  - configurable `maxSackBlocks_` parameter

Two-Way TCP ("FullTCP")

- most TCP objects are one-way (and require a source/sink pair)
- real TCP can be bi-directional
- simultaneous two-way data transfer alters TCP dynamics considerably
- (new– still undergoing debugging)
- the `TCP/FullTcp` agent:
  - follows closely to "Reno" TCP implementation in 4.4 BSD
  - byte-oriented transfers
  - two-way data supported
  - most of the connection establishment/teardown
  - symmetric: only one agent type used for both sides

FullTCP Parameters

- Parameters and defaults:

```
Agent/TCP/FullTcp set segsperack_ 1      ;# segs received before generating ACK
Agent/TCP/FullTcp set segsize_ 536      ;# segment size (MSS size for bulk xfers)
Agent/TCP/FullTcp set tcprexmtthresh_ 3   ;# dupACKs thresh to trigger fast rexmt
Agent/TCP/FullTcp set iss_ 0            ;# initial send sequence number
Agent/TCP/FullTcp set nodelay_ false      ;# disable sender-side Nagle algorithm
Agent/TCP/FullTcp set data_on_syn_ false        ;# send data on initial SYN?
Agent/TCP/FullTcp set dupseg_fix_ true    ;# avoid fast rxt due to dup segs+acks
Agent/TCP/FullTcp set dupack_reset_ false  ;# reset dupACK ctr on !0 len data seg
s containing dup ACKs
Agent/TCP/FullTcp set interval_ 0.1           ;# delayed ACK interval
```

Traffic Sources

- Sources ("applications") used to drive agents
- currently used only by TCP
- Types:
  - Telnet - simulates characters typed by a user
  - FTP - bulk data transfer
- OTcl Interface:

```
$src start                  ;# start sending packets
$src stop                   ;# stop sending packets
$src attach-agent           ;# associate agent with source
$ftpsrc produce npkts       ;# send npkts number of packets
$ftpsrc producemore npkts   ;# send npkts more packets
```

- API is still under some development
- sources only used by TCP at this time

Telnet Traffic Source

- may specify *interval*
- if zero, picks randomly among 10000 measured interarrivals (TCPLIB)
- if nonzero, uses scaled exponential for interarrivals
- packet size constant (but available via bind call)

CBR and UDP Agents

- CBR Agents:
  - stands for "constant bit rate"
    (not really used only this way)
  - non-connection-oriented sending agent
  - sends packets at periodic interval or quasi-periodically
  - constant-size packets
- UDP Agents:
  - very similar to CBR agents
  - uses `TrafficGenerator` class for packet sizes/times

RTP and RTCP Agents

- RTP - "Real-time" (transport) protocol (RFC 1889)
  - implemented as `Agent/CBR/RTP` object
  - special "RTP" header (contains seq number and srcID)
  - sends data periodically similar to CBR sources
  - resets faster when moving from high to low rate
- RTCP - control protocol for RTP
  - implemented as `Session/RTP` object
  - sends at rate based on number of other senders
  - reports known sources and stats

Other Simple Agents

- the `LossMonitor` agent:
  - monitors arrivals of packets
  - looks for sequence number holes
  - provides counters for:
    - **`nlost_`** - number of holes in number space
    - **`npkts_`** - packet arrivals
    - **`bytes_`** - byte arrivals
    - **`lastPktTime_`** - time of last arrival
    - **`expected_`** - next seq number expected
- the `Message` agent:
  - very simple agent
  - allows for including text "messages" in packets
  - currently limited to at most 64 byte (short) messages

Tap Agents and the “Real World”

- allows the simulator to interact with a real network (currently experimental)
- Tap Agents:
  - for now uses 1600 byte buffer as “header” (ie. ether frame + slop)
  - bi-directional agent between simulation and network
  - uses abstract “network” object
  - receives one packet per event (handled through Tcl I/O)
- the `Scheduler/RealTime` class
  - special version of (currently List-based) scheduler
  - ties simulated time to real-time
  - for now, punts if simulation gets far behind
  - (can still do interesting things!)

Network Object

- abstraction of a (real-world) network
- base class for specific network types (e.g. IP network)
- used by other tools in Katz/McCanne/Brewer's MASH project (see `http://www-mash.cs.berkeley.edu/mash/index.html`)
- Network class:
  - requires socket system API (UNIX or WinSock)
  - supports a basic send/recv interface
  - separate send/recv "channels" (i.e. sockets)
  - non-blocking optional
  - framework supports multicast, addr/iface selection, etc
- IP Network (`Network/IP` class)
  - multicast group membership
  - loopback on/off control
  - implements multicast and unicast controls for IP networks

Traffic Generator

- generate traffic according to distributions or traces
- generally used for CBR/UDP agents
- Exponential
  - exponentially distributed on/off times
  - parameters: ontime, offtime, rate, packet size
  - what these mean:
    - burst for expo time with mean ontime
    - be silent for expo time with mean offtime
    - while bursting, send at rate rate
    - use appropriate inter-departure time given rate/size
- Pareto
  - pareto distributed on/off times
  - (many aggregated together can be LRD)
  - parameters: ontime, offtime, rate, shape, packet size
  - what these mean:
    - like expo, except pareto using shape parameter

Trace-Based Traffic Generator

- generate traffic according to trace file
- two classes: `Tracefile` and `Traffic/Trace`
- trace file uses small binary format:
  - first 32-bit field: inter-packet time (microsecs)
  - second 32-bit field: packet size (bytes)

Queue Management and Packet Scheduling

- *buffer management:* how to hold and toss (mark) packets
- *packet scheduling:* what packets get to depart when
- Buffer management:
  - Drop-tail (FIFO)
  - Random Early Detection (RED)
- Packet scheduling:
  - FIFO
  - CBQ (includes priority + round-robin)
  - Round-robin (DRR)
  - Variants of FQ (WFQ, SFQ)

Queue Handlers

- Dequeued packets are often sent downstream to *delays*
- delays (usually) cause two actions:
  1. the packet is scheduled to arrive downstream at time $t + d$
  2. the queue becomes unblocked at time t
  3. t is transmit time, d is prop delay time
- so, delays represent a commonly-occurring *scheduling barrier*
- Queue parameters:

```
Queue set limit_ 50        ;# max packet count in queue
Queue set blocked_ false        ;# queue starts off blocked
Queue set unblock_on_resume_ true      ;# queue is unblocked after resume
```

- control of blocking can be useful for queue banks (e.g. CBQ)

Drop Tail and RED Queues

- Drop-Tail Queues (`Queue/DropTail` class)
  - simple FIFO, drop-tail queues
  - drop from tail when occupancy reaches `qlim_`
- RED (Random Early Detection) Queues (`Queue/RED` class)
  - *active* buffer management technique
  - two thresholds: *minth* and *maxth*
  - also a maximum probability *maxprob*
  - compute *average* queue occupancy over time
  - if average exceeds maxth (or `qlim_`) drop a packet
  - if average is under minth, allow packet to enter queue
  - between, scale drop probability linearly on $[0, maxprob]$

RED Queue Parameters

- `bytes_` do computations in bytes instead of packets (requires assignment of a mean packet size estimate)
- `thresh_` - min thresh
- `maxthresh_` - max thresh
- `mean_pktsize_` - used for computing estimated link utilizations during idle periods
- `q_weight_` - weight given to instantaneous queue occupancy for EWMA
- `wait_` - RED should force a wait between drops
- `linterm_` - reciprocal of maxprob
- `setbit_` - mark instead of drop
- `drop-tail_` - drop new pkt instead of random one

Trace and Monitoring Support

- Two main items: *traces* and *monitors*
- Traces - write an entry for some event (often packet arrivals/departures/drops)
  - `Trace/Enque` - a packet arrival (usually at a queue)
  - `Trace/Deque` - a packet departure (usually at a queue)
  - `Trace/Drop` - packet drop (packet delivered to drop-target)
- Monitors - keep statistics about arrivals/departures/drops (and flows)
  - `SnoopQueue/Out` - on output, collect a time/size sample (pass pac ket on)
  - `SnoopQueue/Drop` - on drop, collect a time/size sample (pass pack et on)
  - `SnoopQueue/EDrop` - on an ”early” drop, collect a time/size sampl e (pass packet on)
  - `QueueMonitor` - receive and aggregate collected samples from snoo pers
  - `QueueMonitor/ED` - queue-monitor capable of distinguishing between “early” and standard packet drops
  - `QueueMonitor/ED/Flowmon` - per-flow statistics monitor (manager)
  - `QueueMonitor/ED/Flow` - per-flow statistics container

Trace File Format

- File format for traces generally of this form:

```
+ 1.45176 2 3 tcp 1000 ---- 1 256 769 27 48
+ 1.45276 2 3 tcp 1000 ---- 1 256 769 28 49
- 1.46176 2 3 tcp 1000 ---- 1 256 769 22 43
+ 1.46176 2 3 tcp 1000 ---- 1 256 769 29 50
+ 1.46276 2 3 tcp 1000 ---- 1 256 769 30 51
d 1.46276 2 3 tcp 1000 ---- 1 256 769 30 51
- 1.47176 2 3 tcp 1000 ---- 1 256 769 23 44
+ 1.47176 2 3 tcp 1000 ---- 0 0 768 3 52
+ 1.47276 2 3 tcp 1000 ---- 0 0 768 4 53
d 1.47276 2 3 tcp 1000 ---- 0 0 768 4 53
```

- Fields: arrival/departure/drop, time, trace link endpoints, packet type, size, flags, flow ID, src addr, dst addr, sequence number, uid
- Many of these fields are from the common packet header:

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

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

        /* per-field member functions */
        int& ptype() { return (ptype_); }
        int& uid() { return (uid_); }
        int& size() { return (size_); }
        int& iface() { return (iface_); }
        double& timestamp() { return (ts_); }
};
```

Trace Callbacks

- may opt to invoke a Tcl function in lieu of writing to file
- see the file `tcl/ex/callback_demo.tcl`

```
MyTest instproc begin {} {
        ...
        $link12_ trace-callback $ns_ "$self dofunc"
        ...
}

MyTest instproc dofunc args {
        ... process args ...
}
```

- Args passed to the callback are a string containing a trace output line (e.g.):
  `- 0.80612 0 1 tcp 1000 ------ 0 0.0 1.0 9 13`

Monitors

- Queue monitors: aggregation points for arrival/depart/drop stats
- Flow monitors: similar, but on a per-flow basis
- Snoop queues: part of the topology, "taps" packet flow, delivers samples to associated monitor

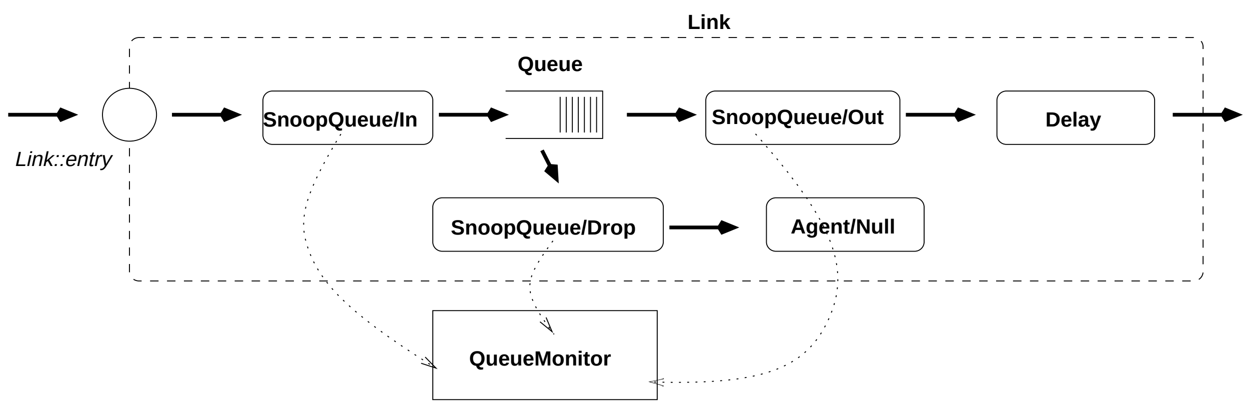


Figure 2: A QueueMonitor and supporting objects

Monitor Stats

- Simple stats kept by monitors:
  - arrivals (bytes and packets)
  - departures (bytes and packets)
  - drops (bytes and packets)
- Aggregate stats (optional):
  - queue occupancy integral
  - (bytes or packets)
- `QueueMonitor/ED` objects
  - "early" drops (bytes and packets)
  - some drops have this distinction (e.g. RED)
- Flow monitors:
  - types `QueueMonitor/ED/Flow` and `QueueMonitor/ED/Flowmon`
  - same as queue monitors, but also on per-flow basis
  - flow defined as combos of (src/dst/flowid)
  - flow mon aggregates and creates new flow objects

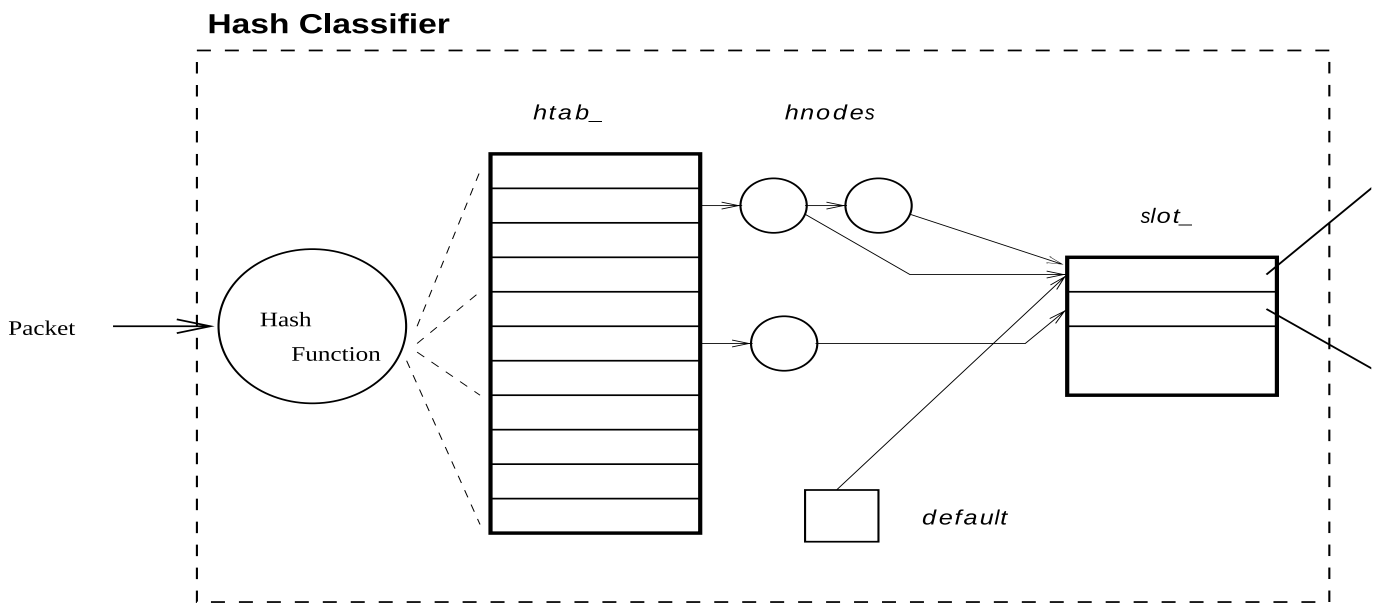
Mathematical Support

- Random number generation
  - RNG implemented in simulator
    (should produce same results on various platforms)
  - based on S. Park and K Miller, CACM 31:10, Oct. 1988
  - support for multiple streams
  - different seeding options
- Random variables
  - distributions applied to RNG streams
  - distributions: uniform, exponential, pareto, constant, hyper-exponential
- Integrals
  - approximation of integral by discrete sums
  - used for average queue size computations
- Samples
  - collect samples
  - provides mean, variance, sum, and count

Break. . .

Hash classifier

- Map packets to associated *flows* or *classes*
- Currently: src/dst, src/dst/fid, fid plus **default**



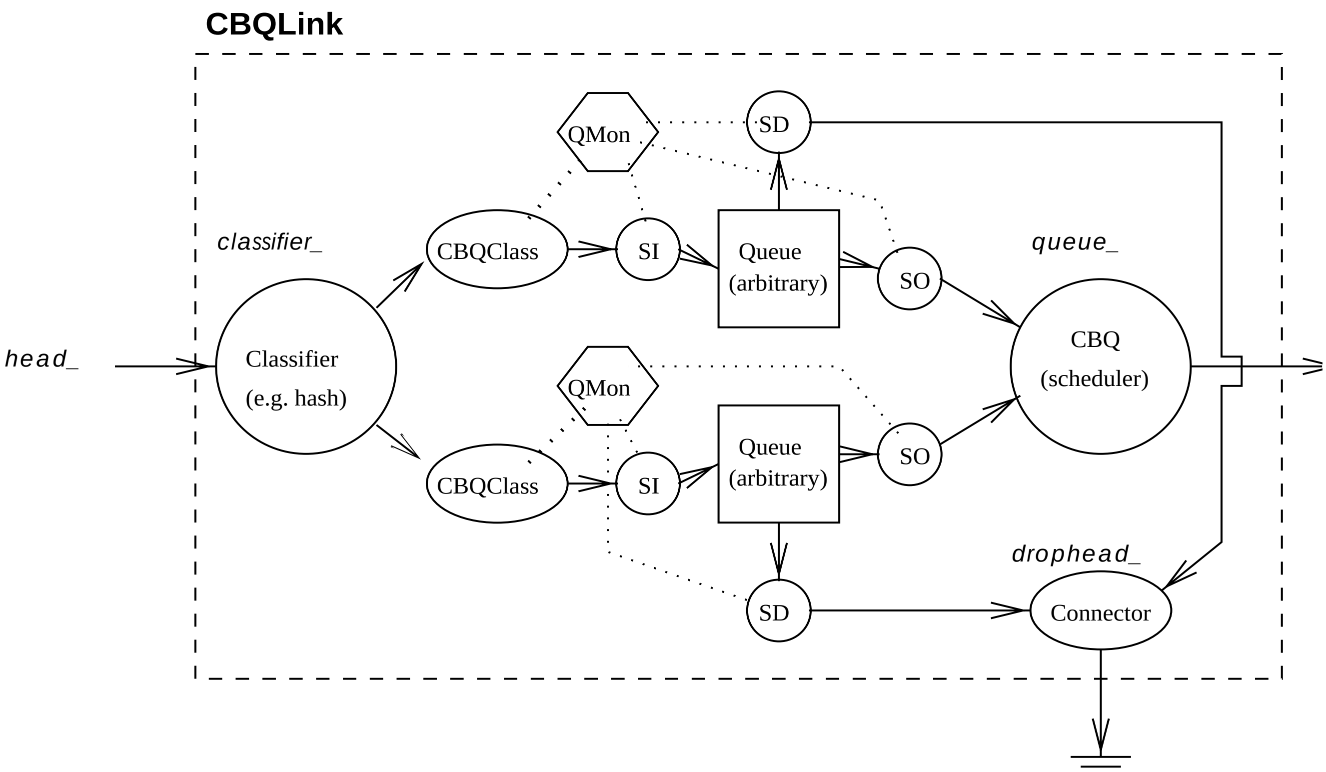
Hash Functions: Source/Dest, Source/Dest/FID, FID

Hnodes: active, slot, src, dst, fid

CBQ: Class Based Queueing

- Floyd and Jacobson, "Link-sharing and Resource Management Models for Packe t Networks", ToN, Aug 1995
- rewrite from CBQ code in ns-1
- packets are members of *classes*
- classes may contain a *priority* and a *bandwidth allocation*
- classes may *borrow* unused bandwidth from other classes
- packets are scheduled using a round-robin scheduler according to the classes they belong to:
  - packet-by-packet RR
  - weighted RR
  - high-to-low priority

CBQ Implementation



- Major components:
  - classifier (maps packets to classes)
  - classes (holds class state)
  - scheduler (schedules packet departures)
- Implemented as a subclass of link: *CBQ link*

Router Mechanisms

- Floyd and Fall, "Router Mechanisms to Support End-to-End Congestion Control", LBNL TR, Feb 1997
- port from ns-1 version based on new FlowMon and CBQ

CBQLink with Router Mechanisms