The 3rd 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
(formerly at USC/ISI)
kannan@catarina.usc.edu
http://www.isi.edu/˜kannan

August 7, 1998

Audience and Outline

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

NSv2 Architecture

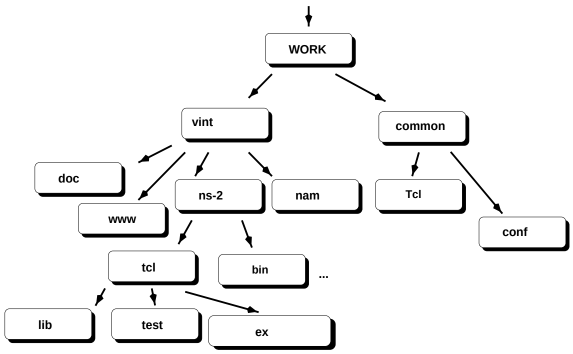
- Object-oriented structure
  - evolution from NSv1 (C++ with regular Tcl)
  - objects implemented in C++ and “OTcl”
  - OTcl: object-oriented extension to Tcl
    (from David Wetherall at MIT/LCS for VuSystem)
    (now supported by UCB Mash group)
- Control/“Data” separation
  - control operations in 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 (fairly steep learning curve)

Development Status

- simulator code basis for VINT Project
- 5ish people actively contributing to the code base
- contributions from: Xerox PARC, USC/ISI, UCB, LBNL
- Some approximate numbers:
  - 44K lines of C++ code
  - 14K lines of OTcl support code
  - 40K lines of test suites, examples
  - 10K lines of documentation!
- Users we know about:
  - 82 universities, 39 companies, 4 US government sites
- 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

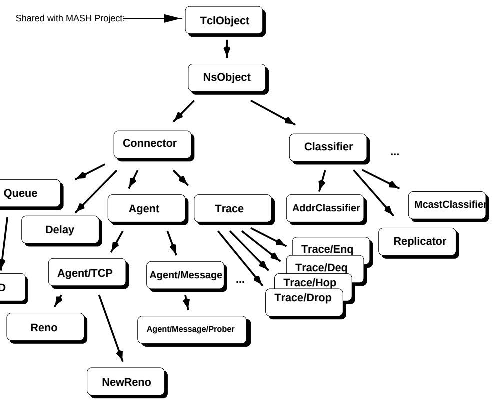
Directory Structure

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



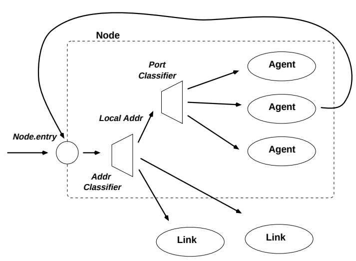
Class Hierarchy

- Top-level classes implement simple abstractions:



Example: a node

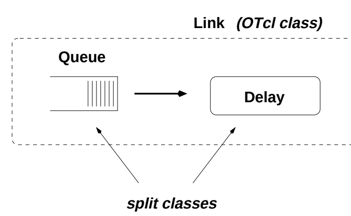
- `Node`: a collection of *agents* and *classifiers*
- Agents: usually protocol endpoints and related objects
- Classifiers: packet demultiplexers



- Note that the node "routes" to itself or to downstream links

Example: a link

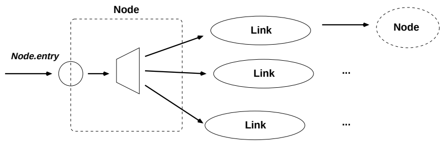
- keeps track of "from" and "to" `Node` objects
- generally encapsulates a queue, delay and possibly ttl checker



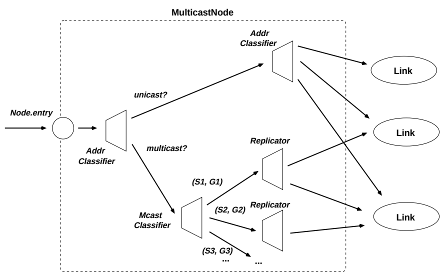
- Many more complex objects built from this base

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)
- For use with tcl8.0, see `http://www-mash.cs.berkeley.edu/dist`

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 using a calendar queue)
- 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
  - `Scheduler/RealTime` - real-time (for emulation)
  - 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 topology 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); }`
- *scheduling barriers*:
  - 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)

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

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

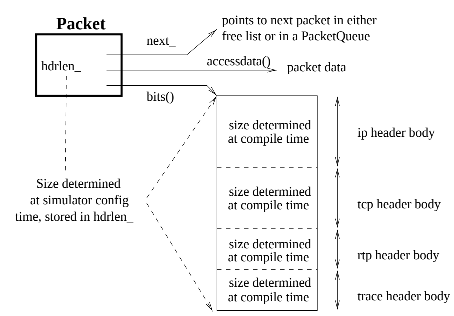
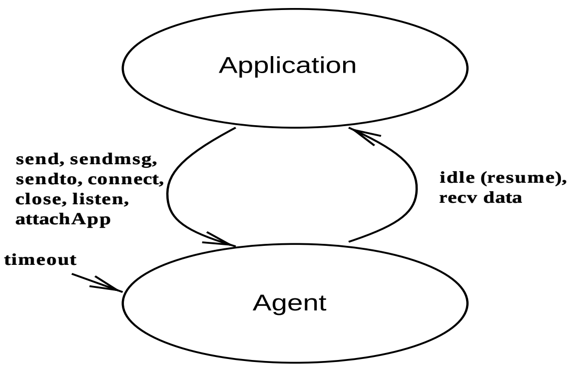


Figure 1: A Packet Object

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

Applications



- *Application:* a model of an application, usually a traffic source
- has an associated *agent*, which usually corresponds to a transport entity
- interface is somewhat like *sockets*

Sources and Traffic Generation

- Applications are of two types: *sources* and *traffic generators*
- Sources are used to drive stream transports (e.g. TCP)
- Traffic generators are used to drive connectionless transports (e.g. UDP)
- Sources
  - Telnet - simulates characters typed by a user
  - FTP - bulk data transfer
- Traffic Generators
  - EXPOO - exponential on/off times, sent at fixed rate
  - POO - pareto on/off times, sent at fixed rate
  - CBR - deterministic rate
  - TrafficTrace - use trace file containing time/len pairs

Sources

- class `Application/Telnet`
  - may specify `interval_` parameter
  - if zero, picks randomly among 10000 measured interarrivals (TCPLIB)
  - if nonzero, uses scaled exponential for interarrivals
  - packet size constant (but available via bind call)
  - Methods:
    - `start` - continuous send
    - `stop` - stop sending
    - `send` n - send n bytes
- class `Application/FTP`
  - bulk data sender used to drive TCP
  - implemented entirely in OTcl
  - Methods:
    - `start` - continuous send
    - `stop` - stop sending
    - `send` n - send n bytes
    - `produce` n - send n packets
    - `producemore` n - send n more packets

Traffic Generation

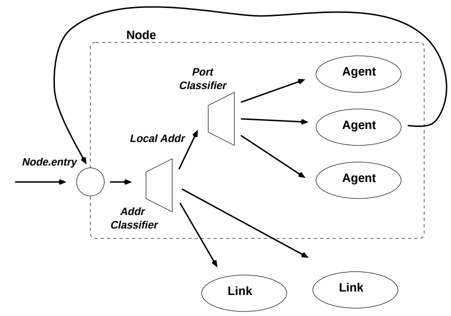
- generate traffic according to distributions or traces
- generally used to drive CBR or UDP transport agents
- Exponential (class `Application/Traffic/Exponential`)
  - exponentially distributed on/off times
  - parameters: ontime, offtime, rate, packet size
  - what these mean:
    - burst for expo 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 (class `Application/Traffic/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 `Application/Traffic/Trace`
- trace file uses small binary format:
  - first 32-bit field: inter-packet time (microsecs)
  - second 32-bit field: packet size (bytes)

Agents

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



- What they provide:
  - a local and destination port address
  - functions for helping to generate/fill-in in packet fields
  - a base class supporting the application interface

Creating a new Agent

- The `Agent` class:

```
class Agent : public Connector {
 public:
        Agent(int pktType);
        virtual ~Agent();
        virtual void timeout(int tno);
        virtual void sendmsg(int nbytes, const char *flags = 0);
        virtual void send(int nbytes) { sendmsg(nbytes); }
        virtual void sendto(int nbytes, const char* flags, nsaddr_t dst);
        virtual void connect(nsaddr_t dst);
        virtual void close();
        virtual void listen();
        virtual void attachApp(Application* app);
        virtual int& size() { return size_; }
        inline nsaddr_t& addr() { return addr_; }

 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 and fill in the API virtual functions
  3. define OTcl linkage functions (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
}
```


CBR and UDP Agent

- CBR Agent:
  - 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 Agent:
  - derived from CBR agent
  - very similar to CBR agents
  - uses `TrafficGenerator` class for packet sizes/times

TCP Agents

- Two categories: one-way and two-way ("full TCP")
- One-way 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"
  - Agent/TCP/Session - shared congestion state w/multiple connections
  - Agent/TCP/Int - per-connection reliability for use w/Session
  - Agent/TCP/*/RBP - Reno, Vegas with Rate Based Pacing
  - Agent/TCP/Asym - TCP mods for asymmetric channels
- One-way TCP receiving agents currently supported are:
  - Agent/TCPSink - one ACK per packet
  - Agent/TCPSink/DelAck - configurable delay per ACK
  - Agent/TCPSink/Sack1 - selective ACK sink (follows RFC2018)
  - Agent/TCPSink/Sack1/DelAck - Sack1 with DelAck
  - Agent/TCPSink/Asym - sink for Asym senders

Two-Way TCP Agents

- Two-way TCP agents (beta test):
  - Agent/TCP/FullTcp - provides Reno functionality
  - Agent/TCP/FullTcp/Tahoe (new)
  - Agent/TCP/FullTcp/Sack (new)
- One-way and two-way TCPs are not interoperable

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)

TCP Agent Parameters

- Common configuration parameters and defaults for TCP agents:

```
Agent/TCP set maxburst_ 0                  ;# max pkts emitted due to 1 recvd
Agent/TCP set maxcwnd_ 0                   ;# max bound on congestion window
Agent/TCP set syn_ false                   ;# do SYN exchange prior to data xfer
Agent/TCP set tcpip_base_hdr_size_ 40      ;# size of TCP/IP hdr, no opts
Agent/TCP set timestamps_ false            ;# due RFC1323-style time stamps
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     ;# do slow-start after idle period
Agent/TCP set tcpTick_ 0.1        ;# timer granularity in sec (.1 is NONSTANDARD)
Agent/TCP set maxrto_ 100000               ;# bound on RTO (seconds)
Agent/TCP set srtt_init_ 0                 ;# initial value for smoothed rtt est
Agent/TCP set rttvar_init_ 12              ;# initial value for rtt var est
Agent/TCP set rtxcur_init 6.0              ;# initial value for current rtx timer
Agent/TCP set T_SRTT_BITS 3                ;# # bits after binary point for SRTT
Agent/TCP set T_RTTVAR_BITS 2              ;# # bits after binary point for RTTVAR
Agent/TCP set rttvar_exp_ 2                ;# exponent of 2 which multiples rttvar
```

- Dynamic values of interest:

```
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
- Asym sinks:
  - ACK pacing

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
- (considered "beta" at this point– requesting feedback)
- 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
- Differences from the "real thing":
  - no receiver's advertised window/persist mode
  - no urgent pointer
  - no 2MSL wait
  - no RST segments
- Now supports Tahoe, NewReno, and Sack variants

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
Agent/TCP/FullTcp set close_on_empty_ false    ;# close conn after send all data
Agent/TCP/FullTcp set ts_option_size_ 10     ;# size of rfc1323 ts option (bytes)
Agent/TCP/FullTcp set reno_fastrecov_ true     ;# congestion window inflation
Agent/TCP/FullTcp set pipectrl_ false      ;# use "pipe" model congestion ctrl
Agent/TCP/FullTcp set open_cwnd_on_pack_ true   ;# increase cwnd on partial acks

Agent/TCP/FullTcp/Newreno set recov_maxburst_ 2    ;# maxburst during recovery

Agent/TCP/FullTcp/Sack set sack_block_size_ 8     ;# # bytes in a SACK block
Agent/TCP/FullTcp/Sack set sack_option_size_ 2        ;# # bytes in opt hdr
Agent/TCP/FullTcp/Sack set max_sack_blocks_ 3      ;# # max # of sack blks
```


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

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_;  // dest for drops
};
```

- if drop target undefined, dropped packets are freed

Introducing Errors

- Packet errors may be introduced into the topology
  - *Error Module* - collection of error models
  - *Error Model* - determines types of errors
- `Error Module` capabilities:
  - insert *classifier* for per-flow error control
  - insert multiple error models
  - collects dropped packets from each error model
  - *default* entry indicates where non-matching traffic goes
- Example:

```
set lossylink_ [$ns link $node_(r1) $node_(k1)]
set em [new ErrorModule Fid]
set errmodel [new ErrorModel/Periodic]
$errmodel unit pkt
$errmodel set offset_ 1.0
$errmodel set period_ 25.0
$lossylink_ errormodule $em
$em insert $errmodel
$em bind $errmodel 0
```


Error Models

- *Error Model:* a parameterized lossy connector (can be used as a base class for other loss models)
- drops packet, sets "error" bit, or ECN indication
- error *units*: packets, bits, time
- base version is associated with random variable or periodic parameters:

```
set errmodel [new ErrorModel]
$errmodel unit pkt
$errmodel set rate_ 0.01
$errmodel ranvar [new RandomVariable/Uniform]
```

- if drop target undefined, sent to module drop target
- Specialized Models:
  - `TwoState` – error and error-free Markov model
  - `MultiState` – arbitrary state transitions in Tcl
  - `Periodic` – sinusoidal (period, phase, burst length)
  - `List` – specifies individual packets/bytes
  - `Select` – like Periodic, but uses packet uid's
  - `SRM` – like Select, but for SRM data only
  - `Trace` – like List, but uses a file
  - `Mroute` – affects certain types of mcast prune messages

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 {
        int     ptype_;         // packet type (see above)
        int     size_;          // simulated packet size
        int     uid_;           // unique id
        int     error_;         // error flag
        double  ts_;            // timestamp: for q-delay measurement
        int     iface_;         // receiving interface (label)
        int     ref_count_;     // free the pkt until count to 0

        static int offset_;     // offset for this header
        inline static int& offset() { return offset_; }
        inline static hdr_cmn* access(Packet* p, int off=-1) {
                return (hdr_cmn*) p->access(off < 0 ? offset_ : off);
        }
        /* per-field member functions */
        ...
};
```


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

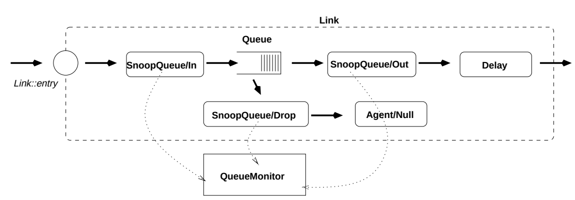


Figure 2: A QueueMonitor and supporting objects

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

Emulation

- allows the simulator to interact with a real network (currently experimental and under development)
- Uses:
  - subject real-world implementations to simulated topologies
  - subject simulated algorithms to real-world traffic
- Traffic transducers:
  - real packets mapped to/from simulated packets via *tap agent*
  - real from either live wire or tcpdump-formatted file via libpcap
- Real-time scheduler
  - 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!)
- Other helpful agents
  - icmp generation
  - ping responder
  - arp responder
  - NAT function

Tap Agents

- Maps simulated packets to/from real-world packets
- Associated `Network` object provides packet source/sink
- Packets are assumed to be network layer, generally IP
- Real traffic is stored in simulated packet's data area:

```
void TapAgent::recvpkt() // net->simulator
{
        ...
        // allocate packet and a data payload
        Packet* p = allocpkt(maxpkt_);
        // fill up payload
        sockaddr addr;  // not really used (yet)
        double tstamp;
        int cc = net_->recv(p->accessdata(), maxpkt_, addr, tstamp);
        // inject into simulator
        hdr_cmn* ch = (hdr_cmn*)p->access(off_cmn_);
        ch->size() = cc;
        ...
        double when = tstamp - now();
        ...
        if (when > 0.0) {
                ch->timestamp() = when;
                Scheduler::instance().schedule(target_, p, when);
        } else {
                ch->timestamp() = now();
                target_->recv(p);
        }
}
```


Network Object

- abstraction of a real-world traffic source/sink
- opened read, write, or read/write by caller
- base class for specific network types (e.g. IP network)
- 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)
  - RAW IP interface (requires privs for raw read/write)
  - multicast group join/leave
  - loopback on/off
  - multicast ttl
- UDP/IP Network (`Network/IP/UDP` class)
  - access to IP/UDP through underlying socket layer
  - does not require privs for use
- Frame Level Packet Filtering and Generation

Packet Capture Network Object

- Use LBNL's `libpcap` facility to access packet traces
- Provides access to trace files and live network packets
- common capabilities:
  - stats (packets captured and dropped)
  - packet filter function (using pcap bpf optimizer)
- Live Packet Capture/Generation (class `Network/Pcap/Live`)
  - promiscuous or ordinary packet capture
  - frame-level packet generation
  - specification of network interface name
- File Packet Capture (class `Network/Pcap/File`)
  - packet capture and filtering only

ARP Module

- not really an *agent* (not derived from `Agent` class)
- uses attached `Network` object for I/O
- provides ARP query/response processing
- also provides proxy ARP
- only works for ethernet now
- (responder is currently under development)
- useful for generating link-layer headers
- ARP cache size configurable by user
- Methods:
  - `network` – set or gets associated Network object
  - `myether` – sets local ethernet address
  - `myip` – sets local IP address
  - `lookup` – looks up mapping for IP address
  - `resolve` – sends ARP request for IP address
  - `insert` – insert an entry into the ARP cache

ICMP Agent

- generates real-world ICMP messages
- class `Agent/IcmpAgent`
- currently supports only ICMP redirect generation
- Interface:

```
set become <who should have sent the redirect>
set target <who to send redirect to>
set dest <redirect for what destination>
set gw <new gateway to use>
$ia send redirect $become $target $dest $gw
```

- provides ability to masquerade as current default gateway (some systems check and require this to process a redirect)
- generates dummy packet (uses defunct GGP protocol) (inspected by victim host to determine which destination modify)

NAT Agent

- supports Network Address Translation (NAT) for use w/emulation
- Currently supports only TCP (UDP is an easy extension)
- Supported modes:
  - source address rewrite
  - destination address rewrite
  - source and destination address rewrite
- does not rewrite port numbers or ACK/sequence numbers

Break. . .

Local Area Networks

- The *lan-link* (subclass of Link) object contains:
  - list of nodes
  - channel object
  - interface queue(s)
  - MAC object(s)
- *Channel* object
  - abstraction of physical layer (PHY)
  - supports contention, collisions and hold/jam
  - simplex or duplex
  - hook for a demux classifier based on MAC id
- *Interface Queues*
  - located between the LL and MAC objects
  - same as Queue objects discussed elsewhere

MAC (Media Access Control) Protocols

- MAC support:
  - MAC addresses
  - CSMA/CA
  - CSMA/CD
  - IEEE 802.11
  - Multihop (e.g. Metricom wireless network)
- Addresses
  - simple integer identifies xmit/recv station
  - multicast support via channel/classifier mods

CSMA-based MAC support

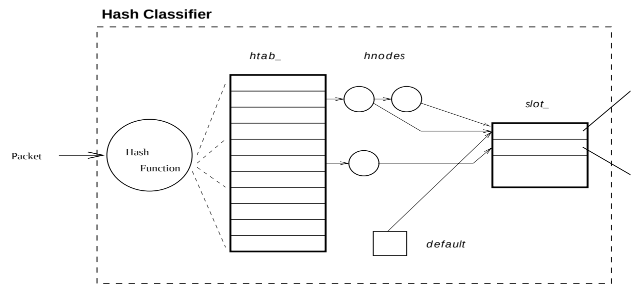
- Common items in CSMA/CA and CSMA/CD:
  - fixed delay/overhead timing
  - inter frame spacing
  - slot time
  - min/max contention window (delay range)
  - retransmission counter and limit
  - carrier sense on/off
  - end-of-contention event handler
  - backoff function
- Differences
  - CSMA/CA - backoff on activity sense
  - CSMA/CD - backoff on collision

IEEE 802.11 Support

- Modes supported:
  - DCF - basic CSMA/CA contention access
  - RTS/CTS - flow control
  - PCF - point coordination
- Parameters:
  - DIFS - Distributed Coordination IFS
  - SIFS - “Short” IFS
  - PIFS - PCF-IFS
  - CSMA/CD - backoff on collision
- See 802.11 standard for more details

Hash classifier

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



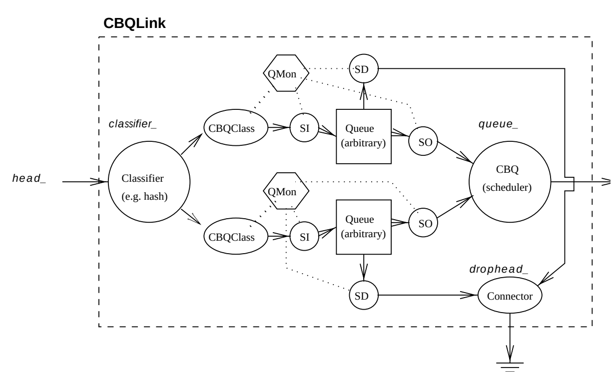
Hash Functions: Source/Dest, Source/Dest/FID, FID

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*

Integrated Services Support

- Developed by Breslau and others at Xerox PARC
- Model:
  1. applications request QoS
  2. network returns yes/no decision
  3. traffic obtains specialized QoS packet scheduling
- Purpose:
  - comparison of admission control schemes
  - (effect on traffic)
  - one model of IETF IntServ Controlled Load (CL)
- Components
  - signalling protocol
  - admission control algorithms
  - QoS-sensitive packet scheduling

IntServ Signalling

- very simple signalling protocol
- (not robust against signalling protocol packet loss)
- operations supported:
  - request, accept, reject, confirm, tear-down
  - extensible to other protocols (e.g. RSVP)
  - one model of IETF IntServ Controlled Load (CL)
- Request includes token bucket params (rate plus depth)
- Confirm is like request, but for existing flow
- decisions are based on admission control algorithms

IntServ Admission Control

- Measurement Based Admission Control (MBAC)
  - accept/reject decision algorithms
  - Measured Sum (Jamin)
  - Hoeffding Bounds (Floyd)
  - Acceptance Region at Origin (Kelly)
  - Acceptance Region Tangent at Peak (Kelly)
  - more coming
- Estimation Algorithms
  - used as input to decision algorithms
  - Time Window
  - Exponential Average
  - Point Sample

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