Using the Open Network Lab

Jon Turner
Applied Research Laboratory
Computer Science and Engineering Department
http://www.arl.wustl.edu/arl

Motivation

■ What is ONL?
  ▷ remotely accessible networking lab
  ▷ gigabit routers with configurable hardware packet forwarding and embedded processors at each port
  ▷ routers can be remotely configured through intuitive GUI
  ▷ extensive support for traffic monitoring/visualization
  ▷ resource for network research community

■ Why did we build ONL?
  ▷ difficult to experiment with high performance routers
  ▷ commercial routers are not open
    ■ open PC routers have limited performance & experiments using them may have limited relevance to high performance routers
  ▷ net research community needs better experimental resources

■ What can you do with ONL?
  ▷ evaluate new and existing protocols & apps in realistic testbed
  ▷ add new features to routers (embedded processors, hw mods)
  ▷ mount compelling demonstrations using real-time visualization
Sample ONL Session

ONL Lab Overview

- Gigabit routers.
  » easily configured thru Remote Lab Interface
  » embedded processors for adding new features
- PCs serve as hosts.
  » half on shared subnets
- Net configuration switch.
  » link routers in virtual topologies
  » traffic generation
- Tools for configuration and collecting results.
  » monitoring traffic
  » data capture and playback
- Open source
  » all hw & sw sources on web
Mitigating Denial of Service Attacks

Users request connections to communicate with the web site.

Extensible router observes partial connections and clears those that don’t complete.

Attacker repeatedly starts the connection process but don’t complete it.

Table fills up blocking legitimate users.

Extensible router requires temporary entry in the table of partial connections.

Target Web Site

Partial Conn. Table

Shadow Table

Extensible Router

Users request connections to communicate with the web site.

Requires temporary entry in the table of partial connections.

Table fills up blocking legitimate users.

Extensible router observes partial connections and clears those that don’t complete.

Attacker repeatedly starts the connection process but don’t complete it.

Connection Table fills when the plugin is off.

Table clears when the plugin is on.

Image xfer blocked.

Image xfer resumes.

Attack Mitigation Displays
People Who Make it Happen

Ken Wong
Lab administration
Web site manager

Jyoti Parwatikar
RLI software
development

Charlie Wiseman
web site dev.
dist. sched.

Fred Kuhns
SPC software
FPX hardware

John Dehart
FPX hardware
System integration

Stephen Levine
plugin development

Gigabit Router Architecture

- Scalable architecture built around ATM switch core.
  - core provides 2 Gb/s bandwidth per port (2x speedup)
- Port processors (PP) implement packet processing
  - Field Programmable Port Extender (FPX) implements routine packet processing
  - Smart Port Card (SPC) hosts programmable extensions
- Control Processor (Linux PC) handles configuration
  - can support routing protocols, OA&M, etc.
ONL Hardware

Field Programmable Port Extender (FPX)

- **Network Interface Device (NID)** routes cells to/from RAD.
- **Reprogrammable Application Device (RAD)** functions:
  - implements core router functions
    - Xilinx Virtex 1E family
      - 38K logic cells (LUT4 + flip flop)
      - 160 block RAMs, 512 bytes each
  - Core router functions include
    - packet classification & route lookup
    - packet storage manager
    - queue manager
      - link queues (datagram, reserved)
      - per flow SPC queues
      - virtual output queues to switch
    - control cell processing
      - access status & control registers
      - update route tables, packet filters

- **SRAM** (1 MB)
- **SDRAM** (128 MB)
- **Reprogrammable App. Device** (38K logic cells +80 KB SRAM)
- **Memory**: 2 Gb/s
- **Network Interface Device**
- **SDRAM** (128 MB)
- **SRAM** (1 MB)
Packet Processing in the FPX

- **Input/Output Segmentation and Reassembly (ISAR/OSAR)**
  - separate reassembly context for link, SPC and each input port
  - IP packets extracted and stored in memory "chunks" by PSM
  - headers passed to "control path"
  - packets retrieved from memory on output and segmented

- **Packet Storage Manager (PSM)**
  - stores packets in one of two SDRAMs based on where arriving from

- **Classification and Route Lookup (CARL)**
  - route lookup (best prefix match) using external SRAM
  - flow lookup (exact 5-tuple match) using external SRAM
  - packet classification (general match) using on-chip resources

- **Queue manager (QM) implements three sets of queues**
  - link queues per-flow and datagram queues using weighted DRR
  - virtual output queues to switch with controllable output rates
    - can be adjusted by control process in SPC
  - SPC queues using weighted DRR

- **Control Cell Processor (CCP)**
  - access to traffic counters, updates to lookup tables & control registers
Classification and Route Lookup (CARL)

- Three lookup tables.
  - Route table for routing datagrams – best prefix
  - Flow table for reserved flows – exact match
  - Filter table for management (adr prefixes, proto, ports)

- Lookup processing.
  - Parallel check of all three
  - Return highest priority primary entry and highest priority auxiliary entry
  - Each filter table entry has assignable priority
  - All flow entries share same priority, same for routes

Route lookup & flow filters
- Share off-chip SRAM
- Limited only by memory size

General filters done on-chip
- Total of 32

Lookup Contents

- Route table – ingress only
  - Output port, Queue Identifier (QID)
  - Packet counter
    - Incremented when entry returned as best match for packet

- Flow table (exact match) – both ingress and egress
  - Output port – for ingress
  - Queue Identifier (QID) – for egress or SPC
  - Packet and byte counters
    - Updated for all matching packets

- Filter table (general) – ingress or egress (rate limits)
  - For highest priority primary filter, returns QID
    - Packet counter incremented only if used
  - Same for highest priority auxiliary filter

- If packet matches both primary and auxiliary entries, copy is made.
Queue Manager

128 flow queues, each with WDRR weight and discard threshold

128 flow queues, each with WDRR weight and discard threshold

64 datagram queues, each with WDRR weight and discard threshold

64 datagram queues, each with WDRR weight and discard threshold

all queues have a byte length that can be queried

Controlling the Queue Manager

- All queues are configurable.
  - discard threshold
  - WDRR quota
- Virtual Output Queues (QIDs 504-511)
  - all packets going to switch placed in VOQ for target output
- Datagram output queues (QIDs 440-503)
  - packets going to link with no special queue assignment are hashed to one these 64 queues
- Reserved output queues (QIDs 256-439)
- SPC queues (QIDs 0-127, 128-255)
  - assigned in pairs \((q, q+128)\)
  - packets to SPC use 1-127
  - packets returning from SPC, going to link use 128-256
FPX Traffic Counters, Status Info.

- Packet and byte counters are read via control cells
  - returned value includes counter value and timestamp
  - timestamps used by software to compute rates
- Port level packet counters
  - received from/sent to link (2 counters)
  - received from/sent to SPC on ingress/egress side (4 counters)
  - received from/sent to router input/output ports (16 counters)
- Packet drop counters
  - ISAR dropped cell counter
  - ISAR invalid packet counter (CRC failure, etc.)
  - QM dropped packet for link queues, switch queues, SPC queues
- And many others,

Selected FPX Counters

00 packets from link
04 packets from ingress port 0
05 packets from ingress port 1
. . .
11 packets from ingress port 7
12 ingress-side packets from SPC
13 egress-side packets from SPC
16 packets to link
20 packets to egress port 0
. . .
27 packets to egress port 7
28 ingress-side packets to SPC
29 egress-side packets to SPC
64 ISAR input cell drops
65 ISAR invalid packet drops
66 QM packet drops for link
67 QM packet drops for switch
68 QM packet drops for SPC
Smart Port Card

- FPGA routes data straight-thru or to/from SPC.
  - 2 Gb/s data paths
- APIC is Network Interface Chip
  - segments packets into cells on transmission
  - reassembles in memory on reception
- 500 MHz Pentium III processor
  - 100 MHz EDO memory
  - 32 bit PCI at 33 MHz
  - flash disk
  - standard BIOS
- Hosts software plugins
  - options processing
  - application-specific processing

Core ATM Switch

- Recycling Buffer
- Virtual Circuit/Path Lookup Table
- 4 Parallel Switch Planes each cell split into 4 pieces
- Resequencing Buffer
- Dual Priority Transmit Buffer
**PVCs for Inter-port Traffic**

- Permanent Virtual Circuits carry traffic between FPXs.
- Egress FPX maintains separate reassembly buffers.
- Can use cell counters in switch to monitor traffic.
- Port-level cell counters also available.

**Switch Congestion**

- **Causes**
  - switch provides bandwidth of about 2 Gb/s per port
  - so, possible for multiple inputs to overload an output causing congestion in switch and lost packets
  - problem can be exacerbated by fragmentation effects
- **Distributed scheduling**
  - distributed scheduler controls VOQ rates to prevent congestion
  - implemented by SPC-resident software modules
    - run periodically (every 500 μs)
    - exchange information on VOQ backlogs, output backlogs
    - adjusts VOQ rate controls in FPX
- **Manual operation**
  - distributed scheduling can be turned off
  - gives user direct control over input VOQ rates
  - makes it possible to overload switch, so use carefully
Coarse-grained nature of scheduling makes it scalable.

Scheduling Algorithm Sketch

- Goals: avoid switch congestion and output queue underflow (work-conservation).
- Backlog proportional sharing.
  - if input accounts for 30% of input-side backlog to output \( j \), it is allocated 30% of switch bandwidth to output \( j \)
- Least occupied output first.
  - each input sends its allocated amount to output with shortest queue
  - as much as it can to output with second smallest queue, etc.
  - continue until all input bandwidth is allocated
- Refinements.
  - add small offset to actual VOQ sizes and use these adjusted VOQ sizes for backlog-proportional sharing
  - second pass to assign otherwise "wasted" input side bandwidth
  - allows packets to flow to previously idle outputs more quickly
- Approximates centralized work-conserving scheduler.
Stress Test

- Inputs combine to build backlogs for outputs 0, 2, 4,…
  - creates input contention
- As inputs “drop out” they switch to unique output.
  - must supply unique output, while clearing backlogs
- If speedup too small, output queues run dry early.
- Stress test can have $p$ phases, $k$ steps per phase
  - ideal output-queued switch can forward all data in $pk$ steps.
  - evaluate scheduler based on its “overshoot”
- Best off-line schedulers require speedup of at least 1.5 for zero overshoot on 16 phase test, with 1 step per phase.

Sample Stress Test Performance

- Scaled down to stretch time scale.
  - 20 Mb/s link rate
  - $S=1.25$
  - 4 phases
  - 3 second phase time

...
Testbed Organization

Control subnet

Internet

ONL server

NetBSD servers for plugin prep

Configuration switch

Major Software Components

Remote Lab Interface (RLI)

SSH proxy

ONL daemon

Switch controller and SPC control message handler

SSH tunnel

NetBSD servers for plugin prep
Getting Started

- Jonathan Turner – 1/31/2006

Getting Started

On-line resources: 
- onl.arl.wustl.edu
- onl.wustl.edu

Get an account

Tutorial

After Logging in

Extra links:
- Getting started
- Status
- Reservations

Install Java runtime environment

Download Remote Lab Interface Software

Configure SSH tunnels

Install JRE (Java Run-time Environment) 1.4.2 (Mandatory)
- Enter java -version if you are running JRE 1.4.2 or higher.
- If you need to install JRE 1.4.2, you can download it from http://java.sun.com/products/1.4.2.

Verify that you can run an telnet client on your host
- Open a command-line window.
- Type telnet hostname port_number, where hostname is your host name and port_number is the port number for the remote server.
- If this command succeeds, you can go on to the next step. Otherwise, continue...
- Open a command-line window.
- Type telnet hostname port_number
- This should display the version number 2 and not the version number 1. If it shows version 1, it should indicate that you are using an older version of the telnet client.

Use an SSH client (e.g., SecureCRT, PuTTY) to connect to the remote server.

Follow these steps:
- Get the RLI file
- Install the Remote Lab Interface (RLI) software
- Configure SSH tunnels
- Test the connection
- Configure the remote lab
- Use the remote lab

Download Remote Lab Interface Software

Configure SSH tunnels
SSH Tunnel Configuration

Name=onl, ports 7070, type=TCP

Configuring Topology

Cluster includes router, GE switch and fixed set of hosts

Add hosts can as needed.

Port 0 used for Control Processor. Spin handle rotates ports.

Drag graphic elements to prettify display.
Add links as needed. These are implemented using configuration switch.

Select "Commit" item to transfer configuration changes to hardware. Note: first time is slow.

Note color change following commit. Indicates RLI connected to lab hw.

Save config. to a file for use in later session.

Right-click on host to get host name and IP address.

n2p2 is local symbolic name 192.168.2.48 is local IP address onl02.arl.wustl.edu is external name.
Verifying Host Configuration

- SSH to host of interest from onl03
- Ifconfig atm0 displays info on host’s ATM interfaces (used by directly connected hosts)
- Verify that IP address of interface matches displayed address.

Configuring Routes

- Click on port to access route table (and other stuff).
- Generate default routes to enable communication.
- Entry defined by address prefix and mask. Specifies router output port.
- To reach other router use port 6.
What does This Mean for Router?

192.168.1.16/28 ⇒ 0
192.168.1.32/28 ⇒ 1
192.168.1.48/28 ⇒ 2
192.168.1.64/28 ⇒ 3
192.168.1.80/28 ⇒ 4
192.168.1.96/28 ⇒ 5
192.168.2.0/24 ⇒ 6

Route table implemented using space-efficient variant of multibit trie.

Verifying Routes

secure shell session to onl02.arl.wustl.edu

ping packets passing through ONL routers
Monitoring Traffic

specify monitoring
select desired monitor variable
select polling rate
peak per ping packet
ping traffic

Adding More Data

forward and return traffic
click on labels to change text
ping traffic
### Changing Routes

Route traffic through bottom link.

- Note effect of route change.

### Using Flow Tables (Exact Match)

- Add filter for port 2.
- Select ingress filter tables.
- Enter 1 for protocol (ICMP).
- Port numbers ignored for ICMP.
- Priority allows flow table entry to override route.
- Traffic switches back to top link.
- Specifies top link.
Using General Filter Tables

- Protocols and ranges may be "don't-care".
- Bottom link addresses may be specified as prefixes.
- Priority allows filter table entry to override flow table entry.
- Traffic switches back to bottom.

Using Auxiliary Filter to Copy Flow

- Double the return traffic.
- Forward traffic on both links.
- Auxiliary filter replicates data stream.
Lab Exercise 1

- Log into ONL web site (as hotI1, hotI2, hotI3, hotI4).
  - view current reservations, status, download RLI

- Run RLI.
  - open SSH connection to onl03.arl.wustl.edu, configure onl tunnel
  - setup configuration with one cluster, extra hosts on ports 4,5
    and link joining ports 6,7
  - generate default routes, commit and save
  - from onl03, open SSH connection to n1p2 and ping other hosts

- Modify routing table and monitor traffic.
  - monitor traffic going both ways on 6-to-7 link
  - start ping from n1p2 to n1p3
  - re-route all traffic from port 2 to port 6
  - re-route all traffic from port 3 to port 7
  - commit

- Experiment with filters.

---

Generating Traffic with Iperf

- Sample uses
  - iperf -s -u
    - run as UDP server on port 5001
  - iperf -c server -u -b 20m -t 300
    - run as client sending UDP packets to server at 20 Mb/s for 300 secs.
  - iperf -s -w 4m
    - run as TCP server on port 5001
    - set max window to 4 MB
  - iperf -c server -w 4m -t 300
    - run as client, sending as fast as possible, with max window 4 MB

- available at http://dast.nlanr.net/projects/iperf/ installed on all onl hosts
Using Iperf

Start UDP receiver

Multiple Iperf Streams

Start UDP sender

Sending bandwidth

Received bandwidth
Iperf Scripts

runServers starts iperf servers remotely on specified hosts.

runClients runs iperf clients in loop with delays to offset start times.

Monitoring Separate Flows

select port-to-port bandwidth.

bandwidths entering and exiting bottleneck link.

desired output.
Incremental Bandwidth Display

- Select Add Formula
- Name: 2.3 to 2.6
- Resulting formula
- Select measures for inclusion
- Name: curve
- Resulting formula

Modifying Link Rate

- Select Queue Tables
- Modify link bandwidth and commit
- Total received bandwidth limited
Mapping Flows to Single Queue

- Packets from each source mapped to common reserved flow queue
- Monitor number of times filters accessed
- Add queue to display using Edit ⇒ Add Queue

Monitoring Queue Length

- Select Egress Qlength
- Queue backlog when two or more active flows
Changing Queue Size

- Change discard threshold and commit larger effective buffer.

Monitoring Discard Rate

- Select FPX general counter.
- Specify counter number.
Mapping Flows to Separate Queues

specify different queues in filters
varying queue lengths
more consistent bandwidth sharing

Changing Bandwidth Shares

proportional bandwidth allocation
vary WDRR quantum
affects packet discard rates
Using Exact Match Filters

- Use `netstat` to determine port numbers.
- Takes priority over general filter.
- Exact match filter to different queue.
- New queue used.

Using Iperf with TCP

- Start TCP sender.
- Start TCP receiver.
- Uses available link bandwidth.
- Queue level responds to rate adjustments.
Iperf TCP Scripts

specify large max window size for high rate flows

% m runTCPservers
#1 /bin/sh

ssh on1$i1.arl.wustl.edu /usr/local/bin/iperf -s -w 4M &
ssh on1$i2.arl.wustl.edu /usr/local/bin/iperf -s -w 4M &
ssh on1$i3.arl.wustl.edu /usr/local/bin/iperf -s -w 4M &
% m runTCPclients
#1 /bin/sh

while true : do
  ssh on1$i1.arl.wustl.edu /usr/local/bin/iperf -c n2p2 -w 4M -t 60 &
  sleep 10
  ssh on1$i2.arl.wustl.edu /usr/local/bin/iperf -c n2p2 -w 4M -t 60 &
  sleep 10
  ssh on1$i3.arl.wustl.edu /usr/local/bin/iperf -c n2p2 -w 4M -t 60 &
  sleep 100
done
%
%

Competing TCP Flows

senders adjust rate to match available bandwidth

per flow queues respond to changes in sending rates
Lab Exercise 2

- Use single cluster configuration with loopback
- Run iperf
  - route all traffic from hosts through port 6
  - create scripts for iperf/udp for 3 flows (1a⇒4, 2⇒5, 3⇒1b)
  - monitor bandwidth on flows entering/leaving bottleneck link
  - monitor packet losses at bottleneck link
  - observe effect of changing link rate
  - modify datagram queue length
- Mapping flows to separate queues
  - use general match filters to map flows entering port 6 to three separate link queues
  - monitor queue lengths
  - run iperf script and observe the queues
  - experiment with discard threshold and WDRR quantum
- Save configuration.
Adding Features with SPC Plugins

SPC uses qid to direct packet to plugin

plugins are kernel-resident software modules

on egress, packets mapped to per-flow queue (128+SPC qid)

filters used to direct packets to SPC queue

on ingress, packets mapped to VOQ

Adding SPC Plugins

pre-defined plugins with numerical identifier

plugin handles packets sent thru FPX queue 8

outgoing link queue 136 = 8 + 128

filter directs packets to SPC queue 8
Effect on TCP

50 ms baseline delay from plugin

delay increases as queue grows

second queue growth period

longer congestion control cycle

performance of delayed flow suffers

Sending Messages to Plugins

Each plugin type implements primitive command interface. Accepts command codes with parameters.

with plugin selected, choose send command item

For delay plugin, command 2 means “change delay” and parameter is new delay value (in ms)
Writing Your Own Plugins

- Plugins are software modules that live within SPC kernel (netBSD).
- Plugins written in C but follow OO-like pattern.
  - plugin type is called a class – each class has a name and numerical id
  - a plugin class must be “loaded” into an SPC before it can be run
  - a class can be instantiated one or more times in an SPC
    - each instance is bound to a queue id, so it can receive packets from FPX
    - each instance may have private data that is retained across packets.
    - may also define class data that is accessible to all instances
- Each plugin class defines a standard set of functions that can be invoked by the plugin environment.
  - pluginName_handle_packet – receive packet and optionally return packet(s)
  - pluginName_handle_msg – receive and respond to control messages
  - pluginName_create_instance – used to initialize per instance variables
  - pluginName_free_instance – used to cleanup data structures
  - miscellaneous other functions – typically don’t require changes
Recipe for Writing a Plugin

- Pick a name (myCounter) and an id (725).
- On ONL user account, create plugins directory with sub-directory for each plugin named in standard way (myCounter-725).
- Copy source code for an existing plugin into new plugin directory.
- Rename the source files to match your plugin.
- In the .h file, find and replace the numerical plugin id.
- In all source files, replace all occurrences of string defining old plugin name with new plugin name (global search-and-replace).
- Modify source code
  - in .h file, add declarations for per instance variables
  - in myCounter_create_instance, initialize per instance variables
  - in myCounter_handle_packet, add code to be executed for received packets
  - in myCounter_handle_msg, add code to implement control messages
- Login to onlbsd1 & compile plugin to object file called combined.o.
- Load plugin onto desired SPC using RLI, install filter and test.

myCounter Plugin Header File

```c
#define myCounter_ID 725;

struct myCounter_instance {
    struct rp_instance rootinstance; // do not touch

    // add declarations for per instance data here
    int count; // number of packets seen so far
    int length; // total length of packets seen
};

void myCounter_init_class();
struct rp_class *myCounter_get_class();
struct rp_instance *myCounter_create_instance(struct rp_class *, u_int32_t);
void myCounter_handle_packet(struct rp_instance *, void *);
void myCounter_free_instance(struct rp_instance *);
void myCounter_bind_instance(struct rp_instance *);
void myCounter_unbind_instance(struct rp_instance *);
int myCounter_handle_msg(struct rp_instance *,
    void *, u_int8_t, u_int8_t, u_int8_t *);
int myCounter (struct lkm_table *, int, int, struct kernel_plugin_fct_struct *);
int myCounter_load(struct lkm_table *, int);
int myCounter_unload(struct lkm_table *, int);
```

myCounter_handle_packet

```c
void myCounter_handle_packet(
    struct rp_instance *this, // pointer to instance structure
    void *bufferList // pointer to list of packet buffers
) {
    struct myCounter_instance
        *inst = (struct myCounter_instance *) this;
    msr_bufhdr_t *buffer = TAILQ_FIRST((HDRQ_t *) bufferList);
    struct ip *iph = msr_pkt_iph(buffer);
    int len = msr_iplen(iph);

    inst->count++;
    inst->length += len;
}
```

- Buffer points to first buffer in list
- Iph points to IP header
- Len is IP packet length in bytes
- Update state

myCounter_handle_msg

```c
int myCounter_handle_msg(
    struct rp_instance *this, // pointer to instance structure
    void *buf, // message as vector of integer values
    u_int8_t flags, // ignore
    u_int8_t seq, // sequence number of message
    u_int8_t *len // number of values in buf
) {
    struct myCounter_instance *inst = (struct myCounter_instance *) this;
    u_int32_t *vals = (u_int32_t *) buf;
    u_int32_t id  = (u_int32_t) htonl(vals[0]);
    u_int32_t typ = (u_int32_t) htonl(vals[1]);

    if (typ == 1) { // return count and length
        vals[0] = (u_int32_t) htonl(inst->count);
        vals[1] = (u_int32_t) htonl(inst->length);
        *len = 2*sizeof(u_int32_t);
    } else if (typ == 2) { // set count and length
        inst->count = ntohl(vals[2]);
        inst->length = ntohl(vals[3]);
        *len = 0;
    }
    return 0;
}
```

- On input,
  - Buf[0]=instance id
  - Buf[1]=msg type
  - Buf[2]=first param
  - Buf[3]=second param
- Type 1: return count, length
- Type 2: set count, length
- Convert between network and host byte order
Passive Monitoring Plugin

- Configure plugin to monitor output traffic at port 6
- Select stats plugin from user directory
- Check auxiliary filter for passive monitoring.

Absorbing Packets

- By default, packet sent to handle_packet are also returned to FPX.
- Stats plugin uses an auxiliary filter to passively monitor traffic, so do not want to return duplicate packets.

```c
void stats_handle_packet(struct rp_instance *this,
                        void *bufferList) {
    struct stats_instance *inst = (struct stats_instance *) this;
    msr_bufhdr_t *buffer = TAILQ_FIRST((HDRQ_t *) bufferList);
    struct ip *iph = msr_pkt_iph(buffer);
    int len = msr_iplen(iph); int proto = msr_ipproto(iph);
    if (proto == 1) {
        inst->icmpCnt++; inst->icmpTot++;
        ...
    }
    // remove packet from input list
    TAILQ_REMOVE((HDRQ_t *) bufferList, buffer, qlist);
    PLUGIN_IPBUF_FREE_FCT(buffer);
}
```
Scripts for Generating Traffic

- start tcp server on one host, udp server on another
- run ping, iperf-tcp and iperf-udp.
- small TCP window to limit bandwidth
- limit UDP bandwidth

Monitoring Plugins

- double-click on plugin name when in monitoring mode
- specify command to send to plugin and returned value to be plotted
Changing Packet Contents

- Can modify packet by simply over-writing.
- StringSub plugin scans for string and replaces it.

```c
void stringSub_handle_packet(struct rp_instance *this, void *bufferList) {
    struct stringSub_instance *inst = (struct stringSub_instance *) this;
    msr_bufhdr_t *buffer = TAILQ_FIRST((HDRQ_t *) bufferList);
    struct ip *iph = msr_pkt_iph(buffer);
    int len = msr_iplen(iph);
    char *p = ((char *) iph) + msr_iphlen(iph);
    char *end = ((char *) iph) + len - 5;
    while (p < end) {
        if (prefix("HELLO", p)) { copy("adieu", p); p += 5; }
        else { p++; }
    }
    // recompute the TCP checksum
    assign_tcpCksums((iphdr_t *) iph);
}
```
Multicast Plugin

- Multicast Plugin configures plugin to make copies for output 3
- Traffic to specified outputs

Making Copies of Packet

- To make copies, must allocate additional buffers and add to buffer list.
- Multicast plugin does this and marks copies so they go to different outputs.

```c
void multicast_handle_packet(struct rp_instance *this, void *bufferList) {
    struct multicast_instance *inst;
    msr_bufhdr_t *buffer, *newBuf;
    MSR_Shim_t *shim, *newShim;
    int i, first, select, len;
    inst = (struct multicast_instance *) this;
    buffer = TAILQ_FIRST((HDRQ_t *) bufferList);
    shim = msr_pkt_shim(buffer);

    if (inst->destVec == 0) {
        // remove packet from input list
        TAILQ_REMOVE((HDRQ_t *) bufferList, buffer, qlist);
        PLUGIN_IPBUF_FREE_FCT(buffer);
        return;
    }
```
len = msr_iplen(msr_pkt_iph(buffer));
first = 1;
for (i = 0, select = 1; i < 8; i++, select <<= 1) {
  if ((inst->destVec & select) == 0) continue;
  if (first == 1) { // don't need to make copy in this case
    first = 0;
    // change value of VOQ in shim to i
    msr_shim_set_ovin(shim, msr_vin_make(i, 0));
    continue;
  }
  // create copy to send to output i
  if ((newBuf = PLUGIN_IPBUF_ALLOC_FCT()) == NULL)
    return; // terminate early if no buffers
  TAILQ_INSERT_TAIL((HDRQ_t *) bufferList, newBuf, qlist);
  // make copy of original packet, including shim
  newShim = msr_pkt_shim(newBuf);
  copy((char *) shim,(char *) newShim, len + MSR_FPX_SHIM_SZ);
  // change value of VOQ in shim of copy to i
  msr_shim_set_ovin(newShim, msr_vin_make(i, 0));
}

SPC Performance – Large Packets

- Iperf packet size is 1470 bytes (31 ATM cells).
- ATM bandwidth of about 200 Mb/s
- About 14,000 packets per second
**SPC Performance – Small Packets**

- Iperf packet size is 52 bytes (2 ATM cells).

- ATM bandwidth of about 115 Mb/s

- about 140,000 packets per second

---

**Debugging Plugins**

- Plugins run in SPC kernel.
  - easy to shoot self in foot
  - limited debugging support

- Helpful tips.
  - start from something that works and make incremental changes
  - store debugging information in state variables and use commands to retrieve values (possibly charting values)
  - for complex plugins, consider creating user-space “testbench” so you can debug in a friendly environment

- Using kernel debugging support.
  - write code
  - compile plugin with debug flag set
  - turn on debugging output using RLI (coming soon)
  - direct debugging output to log file (coming soon)
Useful Pointers

- See online tutorial at onl.wustl.edu.
  - *Router plugins section* has more detailed instructions and examples.
  - *Summary information* section includes:
    - discussion of common problems
    - common Unix commands used for network experiments
    - listing of SPC macros and functions
      - includes packet buffers and header formats
    - list of FPX counters
- System documents.
  - *Design of a High Performance Dynamically Extensible Router* high level system description
  - *Field-Programmable Port Extender (FPX) Support for the Network Services Platform NSP*, details of FPX configurable logic

Sample Uses of ONL

- Study end-to-end performance under controlled conditions.
  - evaluate experimental transport protocols, applications
  - inject cross-traffic, observe low-level behavior using real-time displays
- Add experimental capabilities to routers and evaluate.
  - add plugins to process novel IP options
  - rate reservation, adaptive queue management
  - router assist for large-scale data distribution
  - multimedia services – audio bridging
  - advanced traffic measurement
- Hybrid software/hardware applications.
  - use SPC plugin to modify FPX filters/queues to affect handling of flows
  - SYN attack demo inserts exact match filter for server-to-client data
- Extend hardware capabilities.
  - modify packet scheduling
  - evaluate novel IP lookup or packet classification hardware designs
  - add "sampling" filters to enable SPC for more sophisticated monitoring
Possible Future Extensions

- Improve router functionality.
  - improved link queueing, dynamic packet discards
  - include TCP flags in packet filters
  - sampling filters for netFlow type applications

- Hardware plugin modules.
  - insert hardware processing modules into links
  - implemented using extra FPX modules
  - user-specified FPGA bit-files

- User-specified link delays

- Expanding testbed with NP-based routers.
  - 10 port router implemented with pair of IXP 2850s
  - enable construction of larger networks
  - enable users to more easily modify core router functions
    - queue management, route lookup, packet classification

Lab Exercise 3

- Use single cluster configuration with loopback
- Copy one of the plugins from /users/jst/plugins and implement some extension and test.
  - first, compile and run plugin without change
- For stats plugin, add counter for packets to a specific IP address
  - add command to specify the IP address
  - return value of counter with others
- For stringSub plugin, add commands to change search string and replacement.
  - check for matching lengths or generalize to handle non-matching lengths
- For multicast plugin, change IP destination address in copies.
  - extend command 2 to accept new address as second argument