The Open Network Lab

Ken Wong
Applied Research Laboratory
Computer Science and Engineering Department
http://www.arl.wustl.edu/~kenw
kenw@arl.wustl.edu
http://www.onl.wustl.edu (ONL)
National Science Foundation ANI-023826

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

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

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

People Who Make it Happen

Ken Wong
Admin, Web site, Dist. Sched.
Jyoti Parwatikar
RLI, Software development
Charlie Wiseman
Web site, Ops Dist. Sched.
Fred Kuhns
SPC software
John Dehart
FPX hardware System integration
Stephen Levine
Plugins, Apps.
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

ONL Hardware

- Gigabit Router
- Smart Port Card
- Field Prog. Port Extender

Testbed Organization

- YOU
- Internet
- netBSD servers for plugin prep
- onl server
- onl03
- onlBSD1,2

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.
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 match)*
  - flow table for reserved flows – *(exact match)*
  - filter table for management – *(general match)*
- Lookup processing.
  - parallel check of all three
  - return highest priority primary entry and highest priority auxiliary entry
- General filters done on-chip
  - total of 32

---

Lookup Contents

- Route table *(best match)* – 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 match)* – ingress or egress
  - 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 of pkt is made.
Queue Manager

Egress
- 128 flow queues, each with WDRR weight and discard threshold
- Each queue has WDRR weight and discard threshold
- VOQ per output, each with rate and discard threshold

Ingress
- 64 datagram queues, each with WDRR weight and discard threshold
- All queues have a byte length that can be queried

64 datagram queues, each with WDRR weight and discard threshold

Getting Started
- Get an account
- Tutorial
- Onl.arl.wustl.edu, onl.wustl.edu, www.onl.wustl.edu

Major Software Components

YOU
- Internet
- SSH proxy
- ONL daemon
- Host daemon
- NetBSD servers for plugin prep

Remote Lab Interface (RLI)
- Onl server
- Onl03
- OnlBSD1,2

Configuration switch
- Switch controller and SPC control message handler

Getting Started
- Onl.arl.wustl.edu, onl.wustl.edu, www.onl.wustl.edu

After Logging in
- Extra links
  - Getting started
  - Status
  - Reservations
- Download Remote Lab Interface Software
- Install Java runtime environment
- Configure SSH tunnels
SSH Tunnel Configuration

unix> ssh -L 7070:onl03.arl.wustl.edu:7070 onl03.arl.wustl.edu

Pause for Topology Demo 1/2

- Topology
  » Add Cluster, Add Host, Add Links, Generate Default Routes
  » Spin handle and Port 0
- Loopback

Configuring Topology

Cluster includes router, GE switch and fixed set of hosts

Add hosts can as needed.

Drag icons to improve visual layout

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

Add links as needed. These are implemented using configuration switch.

Select "Commit" item to transfer configuration changes to hardware. Note: first time is slow.
Configuring Topology (cont.)

- 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.
- nsP2 is local symbolic name 192.168.2.48 is local IP address onl02.arl.wustl.edu is external name

Commit (2 NSPs, 4 Extra Hosts)

- Allocates actual hw to logical hw
- Init hosts and NSPs

Internal Interface IP Addresses

- Control Network 128.252.153.X
- 16 x (1+port#) Control Network 128.252.153.X

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

Generate default routes to enable communication.

Entry defined by address prefix and mask. Specifies router output port.

To reach other router use port 6.

Verifying Routes

secure shell session to onl02.arl.wustl.edu

Ping packets passing through ONL routers

Pause for Topology Demo

Unix commands

- ping [-c Count] [-s PktSize] [-i Interval] Target
- netstat -i
- ifconfig {atm0 | eth0 | eth1}

What does This Mean for Router?

Route Table

Flow Table

Filter Table

Input Demux

Result Proc. & Priority Resolution

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

192.168.2.0/24 ⇒ 6

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
192.168.2.32/28 = 0xC0A80220 IP, 0xFFFFFFFF mask
Monitoring Traffic/Real-Time Displays

- Specify monitoring
- Select desired monitor variable
- Select polling rate
- Peak per ping packet
- Ping traffic

Adding More Data

- Click on labels to change text
- Forward and return traffic

Changing Route And Verifying

- Note effect of route change
- Route traffic thru bottom link instead of top link

Built-in Monitoring Points

- Easy to monitor traffic using real-time displays
- Built-in monitoring points
  - Link bandwidth
  - Port-to-port switch bandwidth
  - Queue lengths
  - Packet loss
  - And much more
- Can use same monitoring interface to display data generated by
  - End hosts (e.g., TCP window size)
  - Software plugins installed at router ports
Pause

- Add monitoring display
- Modify route table

Packet Processing Example 1

- Packet Scheduler (PS)
  - Weighted Deficit Round Robin (WDRR) or Distributed Queueing (DQ) on ingress side
  - WDRR on egress side
- Shim is a special header
  - Contains internal metadata

Using Flow Tables (Exact Match)

Using General Filter Tables
Using Auxiliary Filter to Copy Flow

- Forward traffic on both links
- Auxiliary filter replicates data stream
- Double the return traffic

Pause for Filter Demo

- Ingress EM filter (55) to port 7
  \[ (192.168.1.48, 0, 192.168.1.64, 0, \text{proto}=1, \text{voq}=7) \]
- Ingress GM filter (50) to port 3
  \[ (0.0.0.0/0, *, 0.0.0.0/0, *, \text{proto}=*, \text{voq}=3) \]
- Monitor bandwidth
  - Port 7 Egress
- Monitor Egress FPX counters
  - From port 2: Counter 06
  - From port 6: Counter 10

Setup

- two-nsp-300.exp
- urcvrs & yyy &
- usndrs

Study Interaction of Flows

- iperf UDP sndr (250 Mbps each)
- iperf rcvr
- bottleneck (600 Mbps)
- UDP traffic
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

Multiple Iperf Streams

Sending UDP pkts at 250 Mbps from each source ???
- viewed from inside switch
- shim/AAL5 overhead
- `iperf` UDP bw excludes 8-byte hdr

Iperf UDP Scripts

```
#!/bin/sh
source /users/onl/.topology
for host in $n2p2 $n2p3 $n2p4 ; do
  ssh $host /usr/local/bin/iperf -s -u &
done
```

```
#!/bin/sh
source /users/onl/.topology
topology
while true ; do
  ssh $n1p2 /usr/local/bin/iperf -c n2p2 -u -b 250m -t 30 &
  sleep 8
  ssh $n1p3 /usr/local/bin/iperf -c n2p3 -u -b 250m -t 30 &
  sleep 8
  ssh $n1p4 /usr/local/bin/iperf -c n2p4 -u -b 250m -t 30 &
  sleep 10
done
```

Iperf UDP Scripts

```
#!/bin/sh
source /users/onl/.topology
topology
while true ; do
  ssh $n1p2 /usr/local/bin/iperf -c n2p2 -u -b 250m -t 30 &
  sleep 8
  ssh $n1p3 /usr/local/bin/iperf -c n2p3 -u -b 250m -t 30 &
  sleep 8
  ssh $n1p4 /usr/local/bin/iperf -c n2p4 -u -b 250m -t 30 &
  sleep 10
done
```

# Defines $n1p2, $n1p3, $n1p4
Use .topology.csh for csh
Monitoring Separate Flows

- select port-to-port bandwidth
- bandwidths entering and exiting bottleneck link
- desired output

Incremental Bandwidth Display

- select Add Formula
- resulting formula
- select measures for inclusion
- name curve

Modifying Link Rate

- has moved to NSP icon
- 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

Selected FPX Counters

- 00 packets from link
- 04+N packets from ingress port N
- 12 ingress-side packets from SPC
- 13 egress-side packets from SPC
- 16 packets to link
- 20+N packets to egress port N
- 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

See Tutorial => Summary Information => FPX Counters

A-D(RLI): Ingress/Egress => FPX Counters => Ingress/Egress Pkts
Pause

- `two-nsp-300.exp`
  - switch bandwidth = 900 Mbps
  - DQ is ON (versus WDRR voqs)
- `onl03> ~kenw/bin/urcvrs >& log &`
- `onl03> ~kenw/bin/usndrs`
  - client msgs shown on stdout

Mapping Flows to Separate Queues

- Specify different queues in filters
- Vary 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
- New queue used
- Exact match filter to different queue
Using Iperf with TCP

- Ken Wong – 2/21/2006

Iperf TCP Scripts

```bash
#!/bin/sh
source /users/onl/.topology
for host in $n2p2 $n2p3 $n2p4 ; do
  ssh $host /usr/local/bin/iperf -s –w 4M &
done

onl03> trcvrs >& log &
onl03> tsndrs
```

Correction

```bash
#!/bin/sh
source /users/onl/.topology
while true ; do
  ssh $n1p2 /usr/local/bin/iperf -c n2p2 -t 30 &
  sleep 8
  ssh $n1p3 /usr/local/bin/iperf -c n2p3 -t 30 &
  sleep 8
  ssh $n1p4 /usr/local/bin/iperf -c n2p4 -t 30
  sleep 10
done
```

Defines $n1p2, $n1p3, $n1p4
Use .topology.csh for csh

Competing TCP Flows

- Ken Wong – 2/21/2006

Monitoring Host Data

```bash
~kenw/bin/runCwndMon.pl running on $n1p2
Reads /proc/net/tcp
```

Field# starts with 0; space separator

Double click host icon, User Data

Another competing flow started
Setup for Plugin Demo

Recent New Items (Feb 2006)

- Tutorial Pages
  - Examples => Filters, Queues and Bandwidth
  - Examples => Using Router Plugins
- Questions
  - RLI: cse573s@onl.arl.wustl.edu
  - Operations: testbed-ops@onl.arl.wustl.edu
- External host name environment variables
  - ‘source ~onl/.topology’ if using bash
  - Defines environment variables for external host names
  - e.g., $n1p2 might mean onl02.arl.wustl.edu
  - ‘source ~onl/.topology.csh’ if using csh
- Remove need to change modes for monitoring
  - ~onl/export/betaRLI.jar
- Plugin debugging window in RLI
  - Available in both RLI.jar and betaRLI.jar
  - Previously: SSH into CP and run monmsgs

~onl/export/betaRLI.jar

- No need to change modes for monitoring
- Plugin debugging window in RLI
- Negation in GM filters
- User-selectable monitoring value type
  - Open plot label dialogue box
  - Select rate or absolute value
- Distributed Queueing (DQ) is ON by default
  - Other: VOQ rates are settable (600 Mbps default)
    - Default in RLI.jar
  - DQ algorithm avoids input and output overload
    - See SIGCOMM 2004, “Work-Conserving Distributed Schedulers for Terabit Routers” by Pappu, Turner and Wong
    - Tutorial page to come

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
- Lookup
Packet Processing Example 2

- If filter forwards matching pkt to SPC
  - **Ingress**: Output qid is for one of eight VOQs
  - **Egress**: If qid = 8 going to SPC, then output qid = 128+8 = 136
- If you create a new pkt in the SPC (not shown)
  - pkt will be marked as an ingress pkt to be reclassified

Example (2 TCP Flows)

- **Bottleneck**
  - Port 6 egress
  - 100 Mbps
- **Routing (2 flows)**
  - Through port 6
  - n1p2-p3 flow (QID 300) and n1p4-p1a flow (QID 301)
    - Equal quantums
    - 32,000 byte queues
- **Delay plugins**
  - Delay ACK pkts
  - Port 2 egress, 25 msec
  - Port 4 egress, 50 msec

Adding SPC Plugins

- Pre-defined plugins with numerical identifier

Effect on TCP

- Queue 300: n1p2-p3
- Queue 301: n1p4-p1a
- Longer congestion control cycle (50 msec vs 25 msec)
Larger Queues (320,000 Bytes)

- Propagation delay = 50 msec
- Queueing delay = 2 x 24 msec

Sending Messages to Plugins

- For delay plugin, command 2 means "change delay" and parameter is new delay value (in ms)
- Each plugin type implements primitive command interface, handle_msg accepts command codes with parameters.

Plugin Resources

- Tutorial Pages
  - Router Plugins
  - Writing Your First Plugin
- Code Examples
  - ~onl/stdPlugins/
    - COUNTER-432, stats-100, stringSub-102, nullPlugin-103
- Preparation

Writing Your Own Plugins

- Plugins are netBSD kernel modules in SPC
- Written in C but follow OO-like pattern
  - Each plugin class must be "loaded" into an SPC before it can be run
  - Each class can be instantiated one or more times in an SPC
  - Each instance may have private data that is retained across packets.
- Each plugin class has a standard set of functions
  - pluginName_handle_packet - receive pkt; optionally return pkt(s)
  - pluginName_handle_msg - receive and respond to control msgs
  - 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).
- Get the standard files described earlier in “Preparation”
- Run the newplugins.pl script to create the skeleton code
  ```bash
  onl03> cd myplugins
  onl03> ./bin/newplugins.pl myCounter 725
  Makes the directory ~/myplugins/myCounter-725
  Creates files in myCounter-725/ based on files from template directory
  myCounter-725/ (myCounter.hc, stdinc.h, Makefile)
  Alternative: Copy files from some plugin directory and modify names
  and id to suit your needs
- 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_free, add code to implement control messages
- Compile plugin to create combined.o object file: make
  Equivalent to: ssh onlbsd1; cd myplugins/myCounter-725; make; exit
- Load plugin onto desired SPC using RLI, install filter and test.

Standard Plugins (~onl/stdPlugins)

- COUNTER-432
  - Counts and forwards pkts
  - Base plugin
- stats-100
  - Counts ICMP, TCP and UDP pkts and discards
  - Need GM aux filter matching any protocol
- stringSub-102
  - Substitutes “adieu” for “HELLO”
  - Need filter matching TCP pkts
  - Plugin must recompute TCP checksum
- nullPlugin-103
  - Just forward pkts
- multicast-201
  - Multicast pkts to all ports
- psynode-54203
  - SYM flood mitigation plugin
  - Monitors traffic for SYM flood pkts; Installs EM filter in back
    channel; Sends RESET pkts to Web server

myCounter.h 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 *);
int myCounter_handle_msg(struct rp_instance *,
  int, int, int, struct kernel_plugin_fct_struct *);
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 = msr_firstBuffer(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 (include hdr, HBO)

**Packet Buffer**

- **bufferList**
- **msr_pkt_iph()**
- **msr_firstBuffer()**

**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 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) ntohl(vals[0]);
    u_int32_t typ = (u_int32_t) ntohl(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 = 0;
    } 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`
- On output:
  - `buf[0] = count`
  - `buf[1] = length`

**Passive Monitoring Plugin**

- **Forward pktcopy to SPC qid 8**
- **Configure plugin to monitor output traffic at port 6**
- **Check auxiliary filter for passive monitoring.**
- **select stats plugin from user directory**
- **To get to user-defined plugins**
- **Configure plugin to monitor output traffic at port 6**
- **Check auxiliary filter for passive monitoring.**
- **select stats plugin from user directory**
- **To get to user-defined plugins**
Absorbing Packets

- By default, a packet sent to handle_packet is 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 = msr_firstBuffer(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
    msr_removeBuffer(bufferList, buffer);
    msr_freeBuffer(buffer);
}
```

Monitoring Plugins

- Double-click on plugin name when in monitoring mode.
- Specify command to send to plugin and returned value to be plotted.

```
$ onl03> statsClients
```

```
#!/bin/sh
source /users/onl/.topology
while true ; do
    ssh $n1p5 ping –q –i 0.2 –s 1400 –c 200 \\n    n1p1b &
    sleep 8
    ssh $n1p2 iperf -c n1p3 -t 40 &
    sleep 8
    ssh $n1p4 iperf -c n1p1a –u –b 130k -t 40
    sleep 20
done
```

String Substitution Plugin

- Sending file from n2p3 to n2p2.
- Transferred file placed in login directory, with substituted strings.
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 = msr_firstBuffer(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_tcpCksum((iphdr_t *) iph);
}
```

General Rules

- If get copy of pkt for monitoring (aux filter)
  - Remove pkt from buffer list
  - Free buffer memory
  - e.g., stats-100 plugin
- If modify UDP or TCP pkt
  - Recompute checksum
  - e.g., stringSub-102
- If create new pkt
  - Must allocate and fill in buffer and forward pkt
    - Could instead add new pkt to buffer list
  - Default: New pkt is sent to ingress side and reclassified
    - e.g., multicast-201, psyndemo-54203

SPC Performance – Large Packets

- Iperf packet size is 1470 bytes (31 ATM cells).
- ATM bandwidth of about 200 Mb/s
- about 14,000 pps (packets per second)
- 14,000 pps = 7 pkts per 500 usec tick

SPC Performance – Small Packets

- Iperf packet size is 52 bytes (2 ATM cells).
- ATM bandwidth of about 115 Mb/s
- about 140,000 pps (packets per second)
- 140,000 pps = 70 pkts per 500 usec tick
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
  - runs monmsgs on CP host
  - direct debugging output to log file (coming soon)

RLI Plugin Debug Messages

- Debug ON:
  - Run 'monmsgs' on CP
  - Output to /tmp/debug.log
- Direct debugging output to log file (coming soon)

Useful Pointers

- See online tutorial at onl.wustl.edu.
  - More detailed instructions and examples in Router plugins and Examples=>Using Router Plugins sections
  - 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
    - description of some standard plugins
  - Examples section contains recipes for running some examples
- 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
  - System Architecture Document for Gigabit Switching Technology
    - detailed description of core ATM switch

New Features

- Coming Soon
  - Log data to a file
  - Improvements to user data
  - More standard plugins
  - Tutorial => Course Material => Exercises
    - Pedagogic traffic generators
      - udp-echo, udp-sliding-win, etc.
  - Auto-cleaning of /tmp files
  - User-selectable TCP options
- Evaluation Phase
  - Xen environment ➔ Run multiple OSes
    - i.e., different TCP stacks
Possible Future Extensions

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

- Different OSes (and therefore TCP stacks)

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

- Hardware support for user-specified link delays

- Expand 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

Plugin Demo

- Configuration File: one-nsp-delay-ack.exp
  - Rates
    - Bottleneck P6 egress link rate = 100 Mbps
    - P7 egress link rate = 100 Mbps
    - Feeder VOQ rates = 100 Mbps [VOQ(2,6), VOQ(4,6)]
      - Effect: Move bottleneck to feeder VOQs
    - Default 600 Mbps APIC otherwise
  - Queue Sizes
    - eQ(6,300) and eQ(6,301): 32,000 bytes
    - spcQ(2,8) and spcQ(4,8): 640,000 bytes
    - Delay Plugin at egress port 2 (25 msec) and port 4 (50 msec) to delay ACKs
  - GM filters at P6 egress (Qids 300 and 301)

- Shell Scripts
  - delay-rcvrs ($n1p2> delay-rcvrs >& xxx.log)
  - Starts 2 iperf TCP receivers
  - delay-sndrs ($n1p2> delay-sndrs)
  - Starts 2 iperf TCP senders
  - runCwndMon.pl ($n1p2> runCwndMon.pl -f &)
  - Outputs cwnd and ssthresh to ./cwnd-ssthresh.txt file (once/sec)