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 web site.

`Extensible Router`

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

Attackers repeatedly start connection process but don't complete it.

`Partial Conn. Table`

Table fills up blocking legitimate users.

 Requires temporary entry in table of *partial connections*.

`Target Web Site`

`Partial Conn. Table`

`Shadow Table`

`User`

`Attacker`

`Extensible Router`

`Target Web Site`

Attack Mitigation Displays

`Conn. Table fills when plugin off`

`Table clears when plugin on`

`Image xfer blocked`

`Image xfer resumes`
People Who Make it Happen

Ken Wong
Lab administration
Web site manager

Jyoti Parwatikar
RLI software development

Fred Kuhns
SPC software
FPX hardware

John Dehart
FPX hardware
System integration

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.
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

- **External Links**
- **Transmission Interface Cards SPC & FPX underneath**
- **Power Supply**
- **ATM Switch Card at bottom of chassis**
- **SDRAM (128 MB)**
- **SRAM (1 MB)**
- **Reprogrammable App. Device (38K logic cells +80 KB SRAM)**
- **Network Interface Device**

---

9 - Jonathan Turner – 6/13/2005

10 - Jonathan Turner – 6/13/2005
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

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

- Virtual Circuit/Path Lookup Table
- Resequencing Buffer
- Dual Priority Transmit Buffer
- 4 Parallel Switch Planes each cell split into 4 pieces
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, easy for multiple inputs to overload an output causing congestion in switch and lost packets
  - problem can be exacerbated by fragmentation effects

- Congestion avoidance
  - plan experiments to avoid excessive overloads
  - by default, link rates are limited to 600 Mb/s to reduce opportunities for switch congestion

- VOQ rate controls
  - rate limits for virtual output queues can be used to ensure outputs are not overloaded
  - automated configuration of rate limits is planned
    - periodic exchange of VOQ backlog information by SPCs
    - distributed allocation of switch bandwidth
Getting Started

onl.arl.wustl.edu

get an account
tutorial

After Logging in

extra links
getting started
status
reservations

download Remote Lab Interface Software
install Java runtime environment
configure SSH tunnels

Get the ELI jar file
You can do this by either following the Get ELI jar file link or by logging into any host and copying the file ELI.jar from \var\opt\ELI\jar to your remote host.

Install JRE (Java Runtime Environment) 1.4.2 (Mandatory)
- First determine if you can run Java. You can do this by running jre.exe or java.exe and checking if it returns the version of the JRE that you are running.
- If you need to install JRE 1.4.2, you can do so by downloading the JRE from the Sun site and installing it on your system.

Verify that you can run an elbow client on your host
- If your host is running Windows, you can run an elbow client from your command line by running...
- Otherwise, you can...

Get Open Network Laboratory Interface Software
Download the Open Network Laboratory Interface Software from onl.arl.wustl.edu and place it in a directory on your system.

Open ELI Interface Software
Open the ELI Interface Software by double-clicking the ELI.jar file in the directory you placed it in.
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.

Drag graphic elements to prettify display

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 config 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.
Verifying Host Configuration

Directly connected hosts use ATM interface.

Verify that IP address of interface matches displayed address.

/sbin/ifconfig -a displays info on configured interfaces.

Configuring Routes

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

Click on port to access route table (and other stuff).

Default routes can be generated for local hosts.
What does This Mean for Router?

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

Adding More Routes

So traffic carried on top link.

Causes packets received at port 2 for specified host to be routed thru output 6.
Routes for 2-Way Communication

- Commit routing changes to make effective
- First hop of east-bound path
- Second hop of east-bound path
- First hop of west-bound path
- Second hop of west-bound path

Verifying Routes

- Secure shell session to onl19.arl.wustl.edu
- Ping packets passing through ONL routers
Monitoring Traffic

- Specify monitoring view
- Monitor ping traffic
- Select desired monitor variable
- Peak per ping packet

Monitoring Other Data

- To add separate chart
- Enter polling rate
- Packet counter: 0 for packets from link, 16 for packets to link
- Click to change label
- To select FPX packet counter
- Shows packets/sec for entering/exiting traffic
Monitoring Still Other Data

Set focus, so new trace goes here.

Monitor bandwidth use on virtual circuit entering ATM core.

Specify target output port.

New traces from 2 to outputs 6 and 7.

Changing Routes

Commit route change to make effective.

Changing next hop to 7 re-routes flow thru bottom link.

Now see traffic from input 2 to output 7.

No traffic from input 2 to output 6 or east-bound on top link.
Using Flow Tables (Exact Match)

- Select ingress filter tables for port 2
- Priority allows flow table entry to override route
- Traffic switches from port 7 to port 6

To enter:
- Add filter
- Port numbers ignored for ICMP
- Enter 1 for protocol (ICMP)
- Specifies top link
- Protocols and ranges may be "don't-care"

Using General Filter Tables

- Add general filter
- Protocols and ranges may be "don't-care"
- Addresses may be specified as prefixes
- Specifies bottom link
- Priority allows filter table entry to override flow table entry
- Traffic switches back to 7
Using Auxiliary Filter to Copy Flow

<table>
<thead>
<tr>
<th>Exact Match (6)</th>
<th>General Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>source address</td>
<td>source address</td>
</tr>
<tr>
<td>192.168.1.480</td>
<td>192.168.2.480</td>
</tr>
</tbody>
</table>

lower priority irrelevant, since auxiliary filter replicates data stream

flow being sent to both 6 and 7

generating traffic with Iperf

available at http://dast.nlanr.net/projects/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
Using Iperf

45 - Jonathan Turner - 6/13/2005

Multiple Iperf Streams

46 - Jonathan Turner - 6/13/2005
Displaying Incremental Bandwidth

- Select Add Formula
- Name curve
- Select measures for inclusion
- Resulting formula
- Resulting curve

Modifying Link Rate

- Select Queue Tables
- Modify link bandwidth and commit
- Total received bandwidth limited
- Fluctuations due to bursty sources and small default queue sizes
Mapping Flows to Single Queue

- Packets from each source are mapped to a common reserved flow queue.

Monitoring Queue Length

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

- Add Egress Queue
- Delete Egress Queue

### Egress Queues

<table>
<thead>
<tr>
<th>Queue Id</th>
<th>Threshold(Bytes)</th>
<th>Quantum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>84000</td>
<td>2040</td>
</tr>
<tr>
<td>1</td>
<td>120000</td>
<td>2040</td>
</tr>
</tbody>
</table>

- Enter number of queue of interest
- Change discard threshold and commit
- Larger queue smooths out bursts, giving better bw sharing

Mapping Flows to Separate Queues

- Per flow queues ensures fair-sharing
- Vary queue lengths
- Varying queue lengths
Changing Bandwidth Shares

- Vary WDRR quantum
- Proportional bandwidth allocation
- Packet discards

Changing VOQ Rates

- Select Port 2 Queue Tables
- Reduced input rate
- Reduced queuing
Using Exact Match Filters

- Start long iperf session
- Interrupt to run netstat
- Exact match filter to different queue
- New queue used
- Lower priority for general match filter

Using Iperf with TCP

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

- New queue used
- Lower priority for general match filter
Competing TCP Flows

senders adjust rate to match available bandwidth

per flow queues respond to changes in sending rates

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
What Does it Mean?

SPC uses qid to direct packet to plugin

plugins are kernel-resident software modules

returning packets mapped to per-flow queue (128+SPC qid)

filters used to direct packets to SPC queue

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.

For delay plugin, command 2 means "change delay" and parameter is new delay value (in ms).

Effect on TCP

Delayed flow gets fair share of link bandwidth

shorter congestion control cycle
What’s in a Plugin?

- 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;
    mshr_bufhdr_t *buffer = TAILQ_FIRST((HDRQ_t *) bufferList);
    struct ip *iph = mshr_pkt_iph(buffer);
    int len = mshr_iplen(iph);

    inst->count++;
    inst->length += len;
}
```
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) ntohs(vals[0]);
    u_int32_t typ = (u_int32_t) ntohs(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

convert between network and host byte order

Advanced Topics

- Displaying plugin data using RLI.
  » double-click on plugin table entry when in monitoring mode
  » enter number for message to send to plugin
  » enter index of returned value to plot
- Debugging complex plugins.
  » create user-space “testbench” to debug in friendly environment
    • feed packets to handle_packet and observe the results using debugger and/or
      printing debugging output
  » when confident of correctness, test on SPC using debugging mode
    • provides flexible mechanism for sending debugging output to CP for display
    • must compile with plugin with debug flag set
    • turn on debugging output using RLI (coming soon)
    • direct debugging output to log file (coming soon)
- Other SPC kernel mechanisms.
  » registering a callback function – useful for timer-driven operations
  » consuming packets in plugin (for passive monitoring applications)
  » modifying packet data, generating packets
  » dynamically modifying FPX routes/filters/queues
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