Using the Open Network Lab

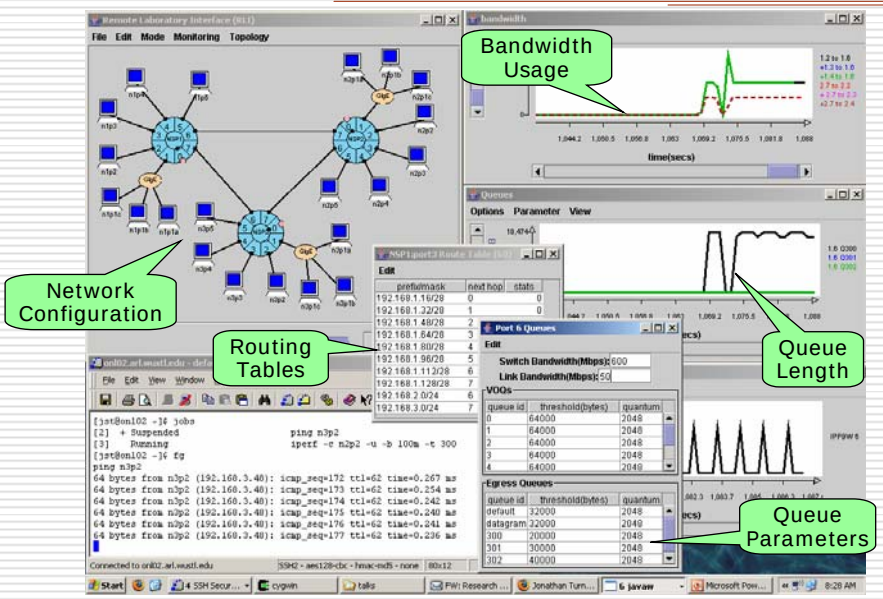
Jon Turner
Applied Research Laboratory
Computer Science and Engineering Department
http://www.arl.wustl.edu/arl

Washington University in St.Louis

Motivation

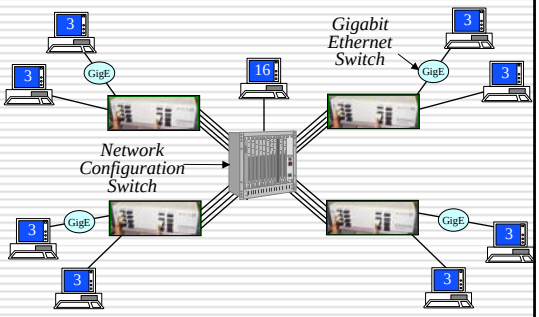
- What is ONL?
  - » remotely accessible networking lab
  - » gigabit routers with configurable hardware packet forwarding and embedded processors at each port
  - » routers can be remotely configured through intuitive GUI
  - » extensive support for traffic monitoring/visualization
  - » resource for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

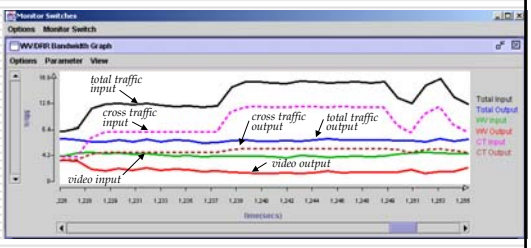
Sample ONL Session



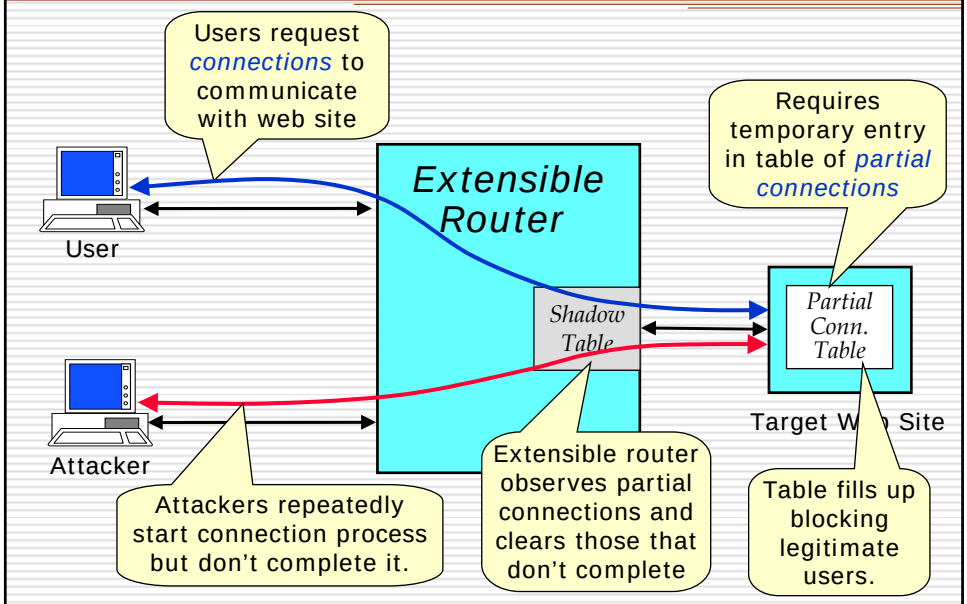
ONL Lab Overview

- 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

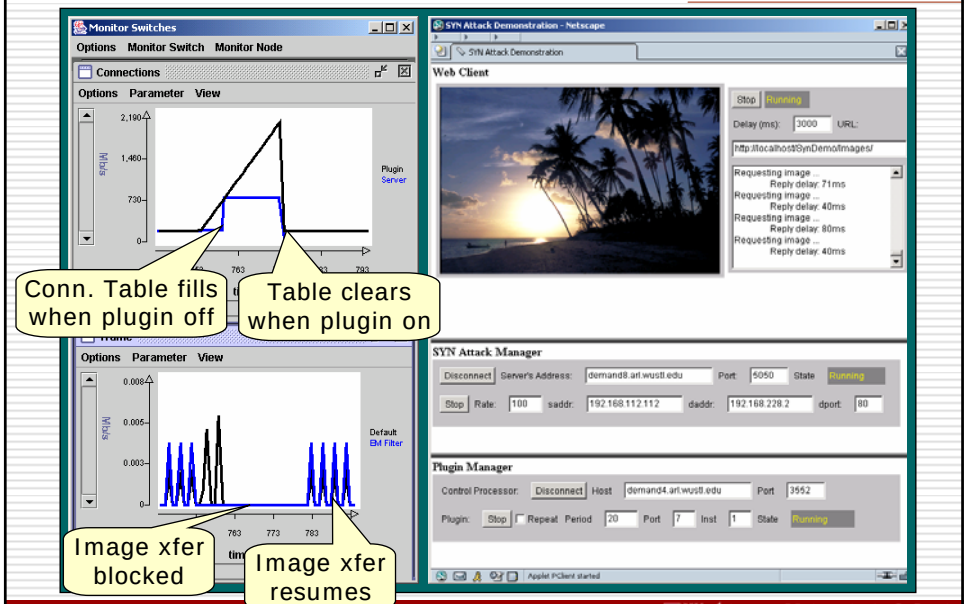




Mitigating Denial of Service Attacks



Attack Mitigation Displays



People Who Make it Happen

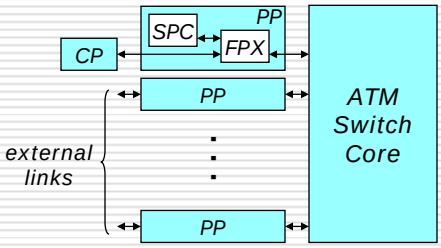
Ken Wong
Lab administration
Web site manager

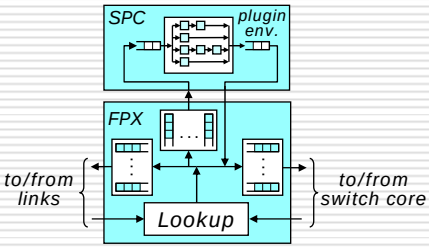
Jyoti Parwatikar
RLI software
development

Fred Kuhns
SPC software
FPX hardware

John Dehart
FPX hardware
System integration

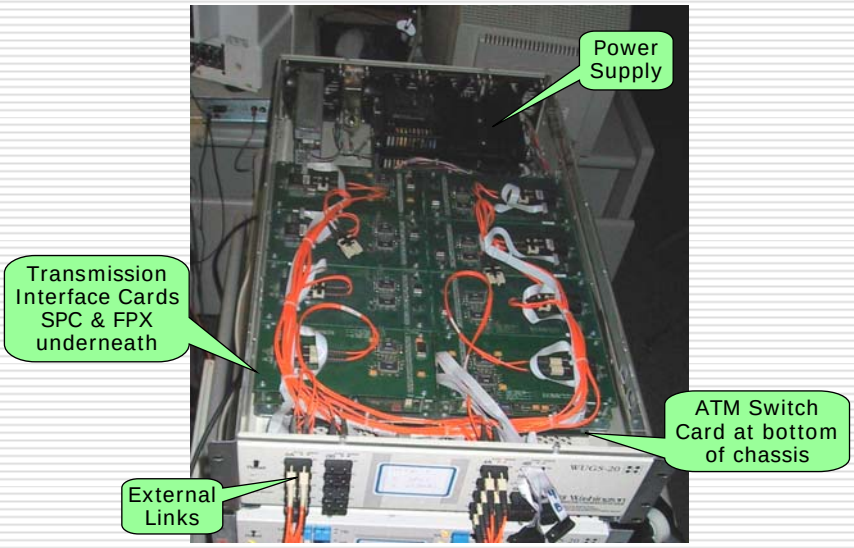
Gigabit Router Architecture





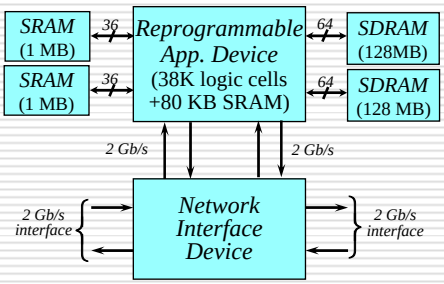
- Scalable architecture built around ATM switch core.
  - » core provides 2 Gb/s bandwidth per port (2x speedup)
- Port processors (PP) implement packet processing
  - » Field Programmable Port Extender (FPX) implements routine packet processing
  - » Smart Port Card (SPC) hosts programmable extensions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

Router Photograph

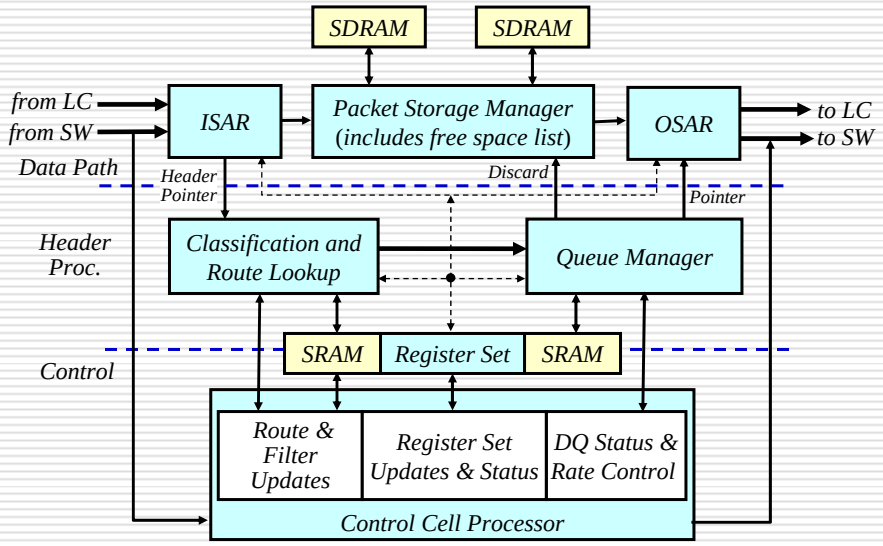


Field Programmable Port Extender (FPX)

- *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



Packet Processing in the FPX

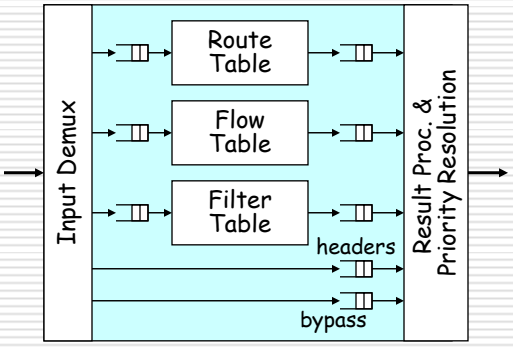


Packet Processing in the FPX

- 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

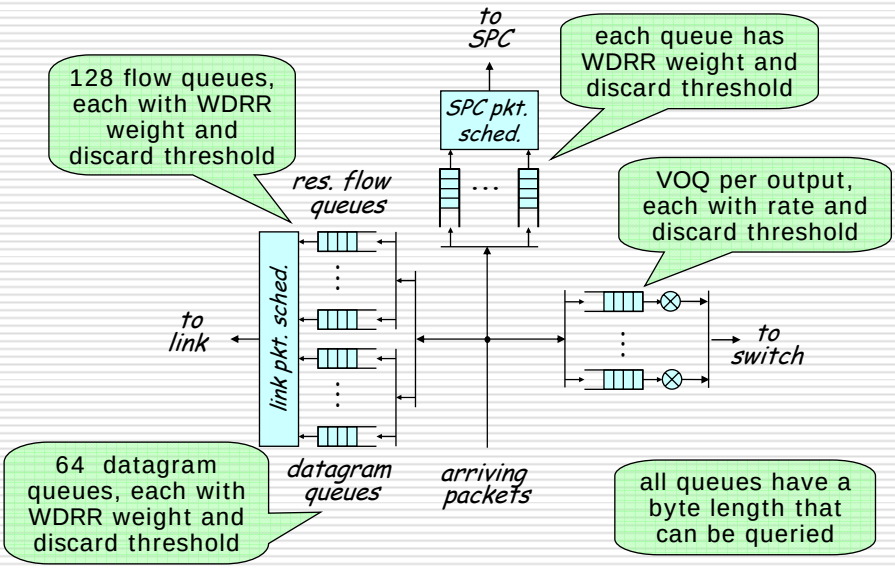


- Route lookup & flow filters
  - » share off-chip SRAM
  - » limited only by memory size
- General filters done on-chip
  - » total of 32

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



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.

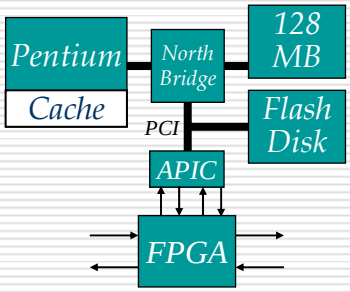
- Packet and byte counters are read via control cells
  - » returned value includes counter value and timestamp
  - » timestamps used by software to compute rates
- Port level packet counters
  - » received from/sent to link (2 counters)
  - » received from/sent to SPC on ingress/egress side (4 counters)
  - » 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

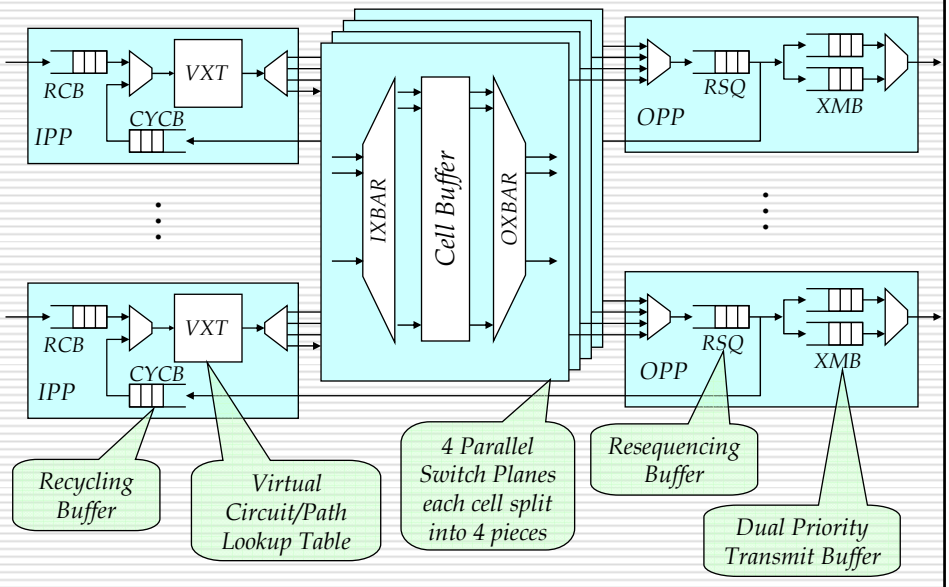
00	packets from link
04	packets from ingress port 0
05	packets from ingress port 1
	. . .
11	packets from ingress port 7
12	ingress-side packets from SPC
13	egress-side packets from SPC
16	packets to link
20	packets to egress port 0
	. . .
27	packets to egress port 7
28	ingress-side packets to SPC
29	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

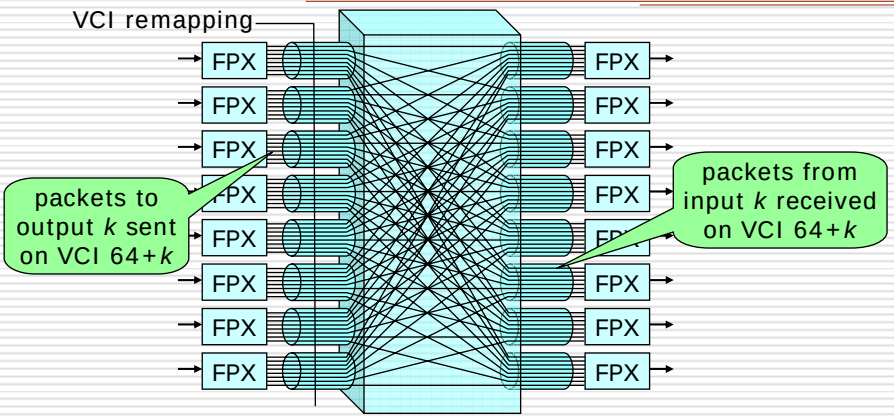
- FPGA routes data straight-thru or to/from SPC.
  - » 2 Gb/s data paths
- APIC is Network Interface Chip
  - » segments packets into cells on transmission
  - » reassembles in memory on reception
- 500 MHz Pentium III processor
  - » 100 MHz EDO memory
  - » 32 bit PCI at 33 MHz
  - » flash disk
  -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



Core ATM Switch



PVCs for Inter-port Traffic

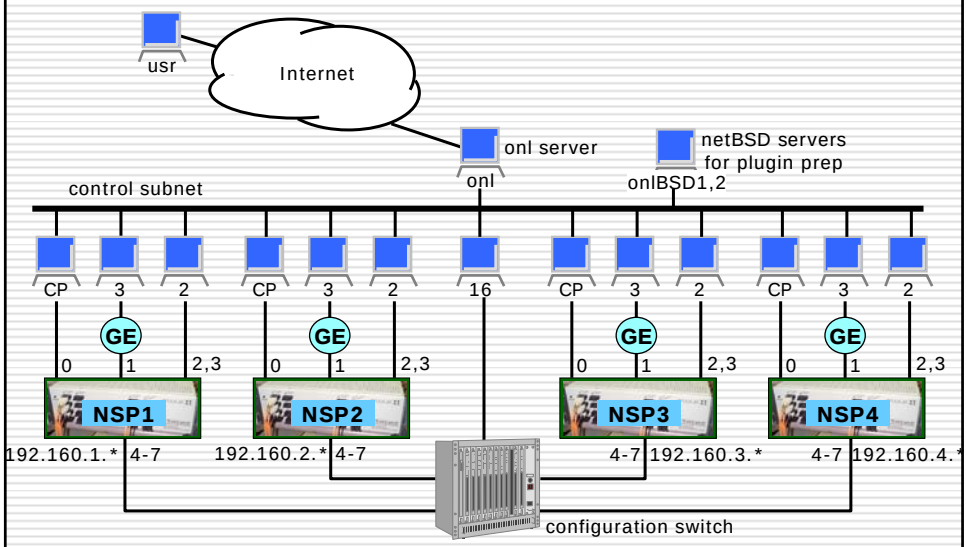


- Permanent Virtual Circuits carry traffic between FPXs.
- Egress FPX maintains separate reassembly buffers.
- Can use cell counters in switch to monitor traffic.
- Port-level cell counters also available.

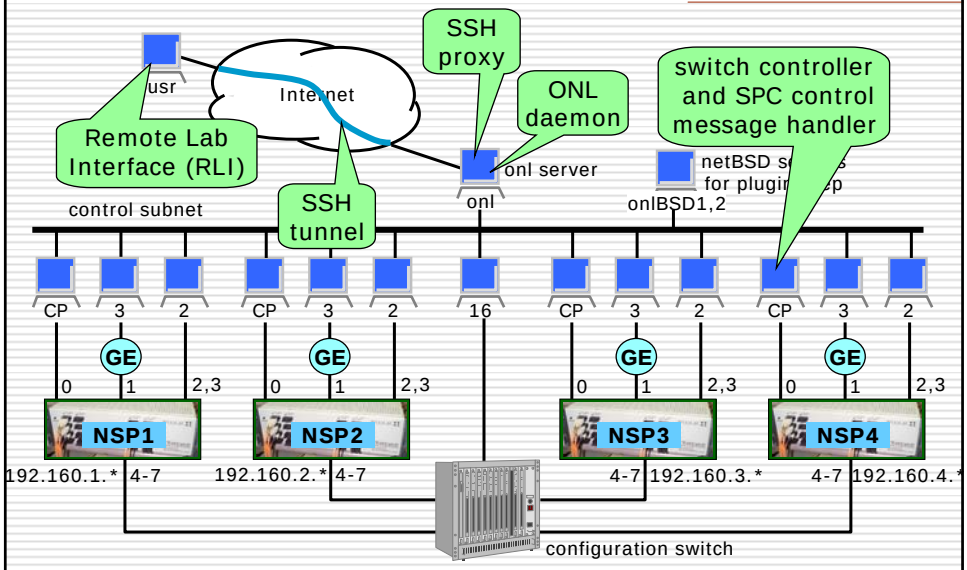
Switch Congestion

- 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

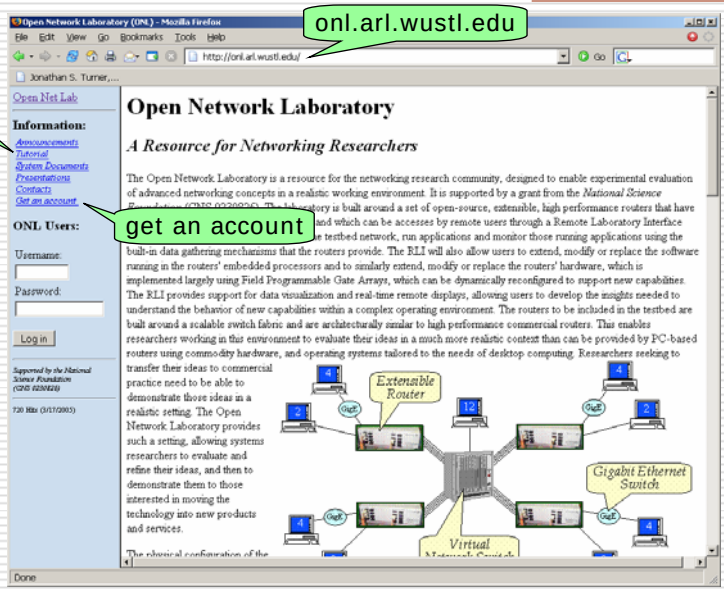
Testbed Organization



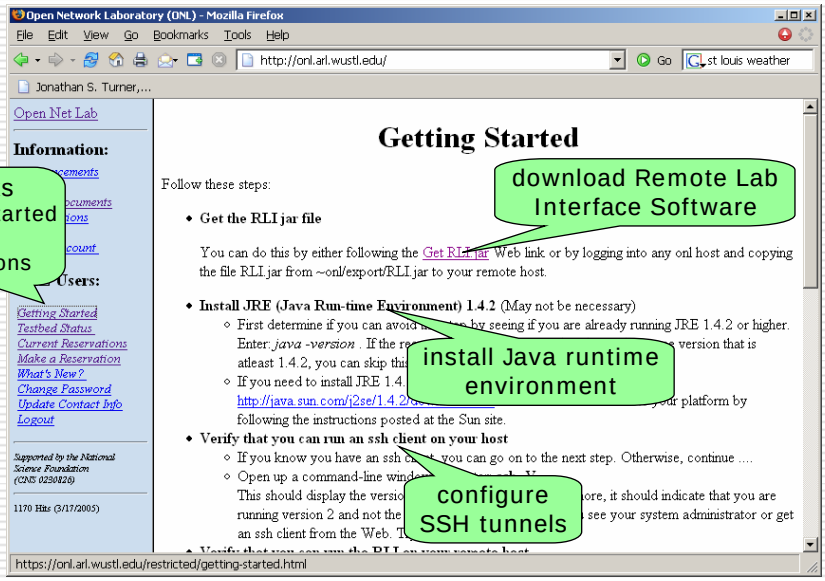
Major Software Components



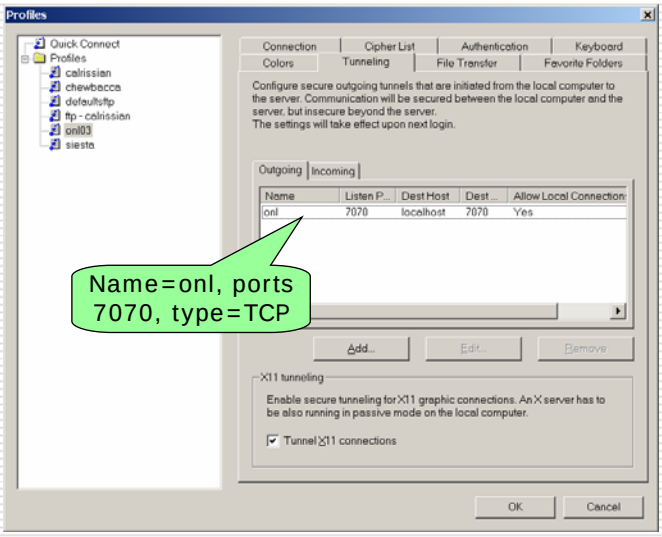
Getting Started



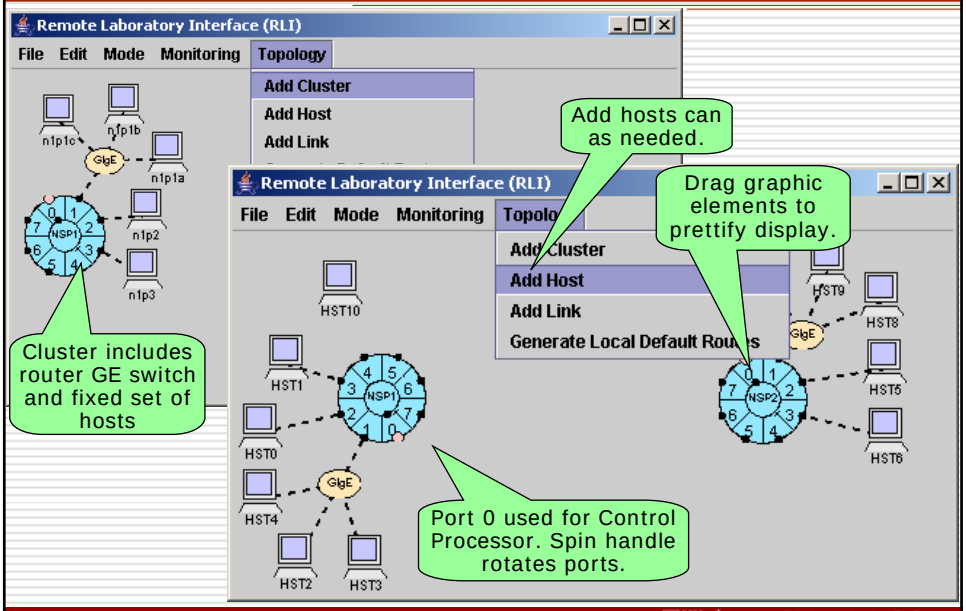
After Logging in



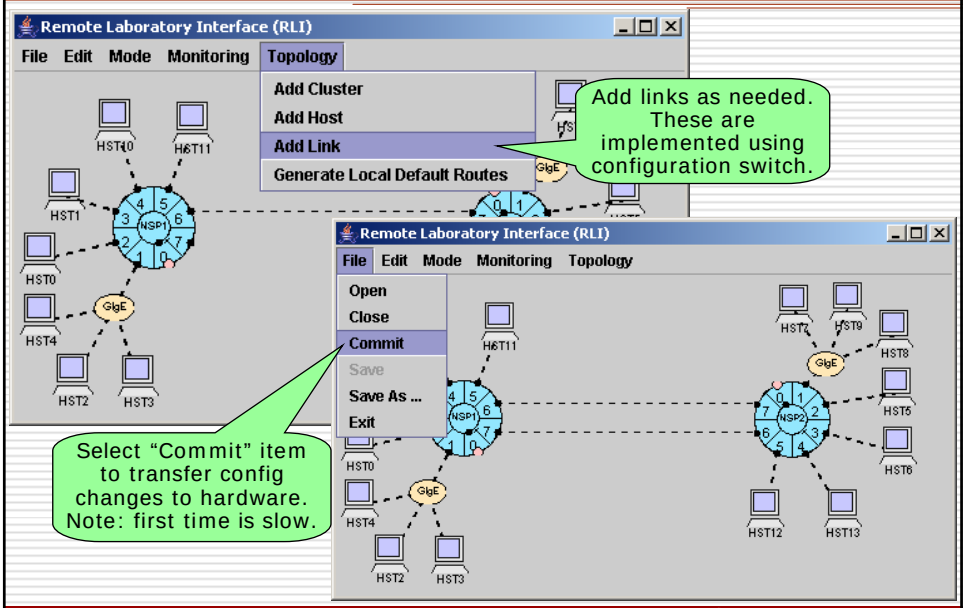
SSH Tunnel Configuration



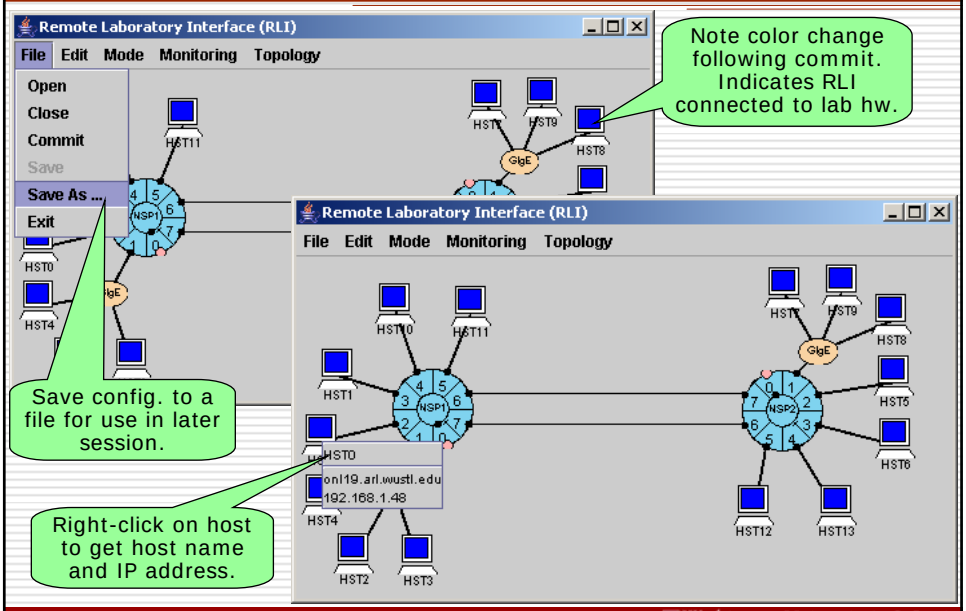
Configuring Topology



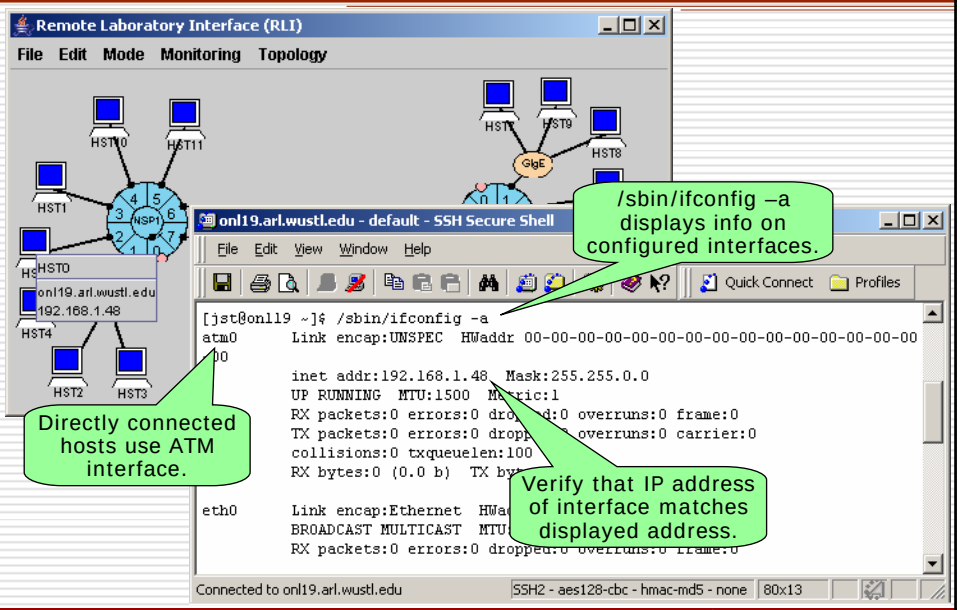
Configuring Topology (cont.)



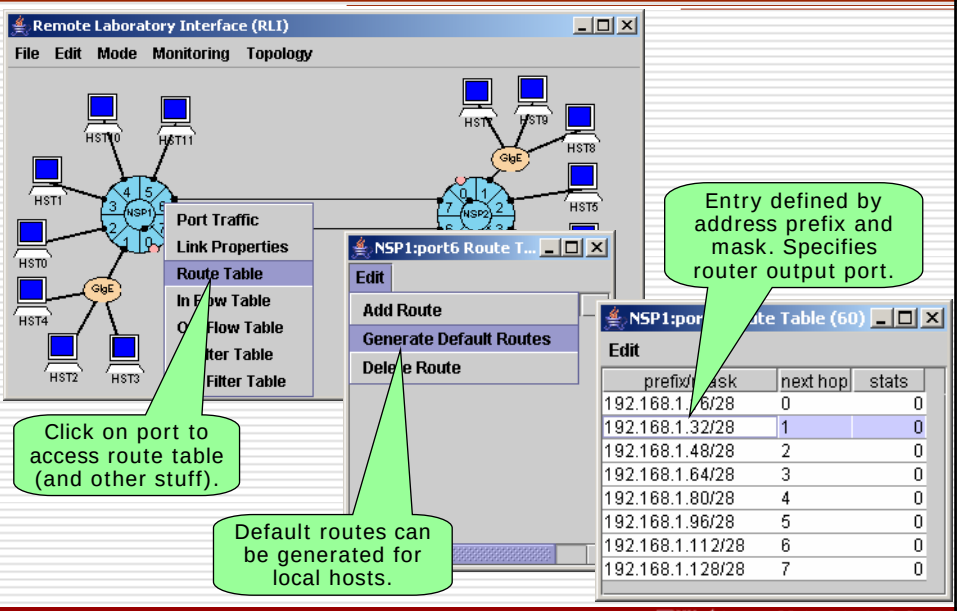
Configuring Topology (cont.)



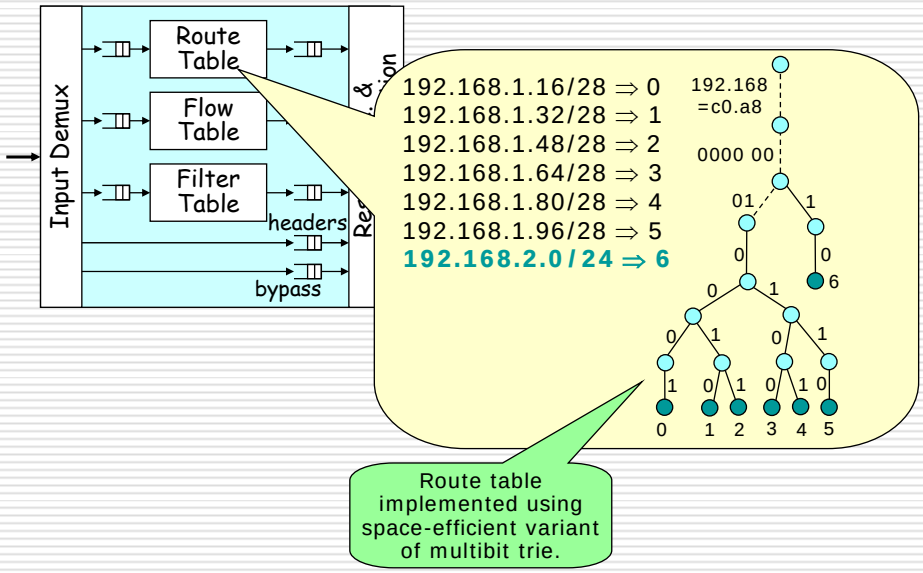
Verifying Host Configuration



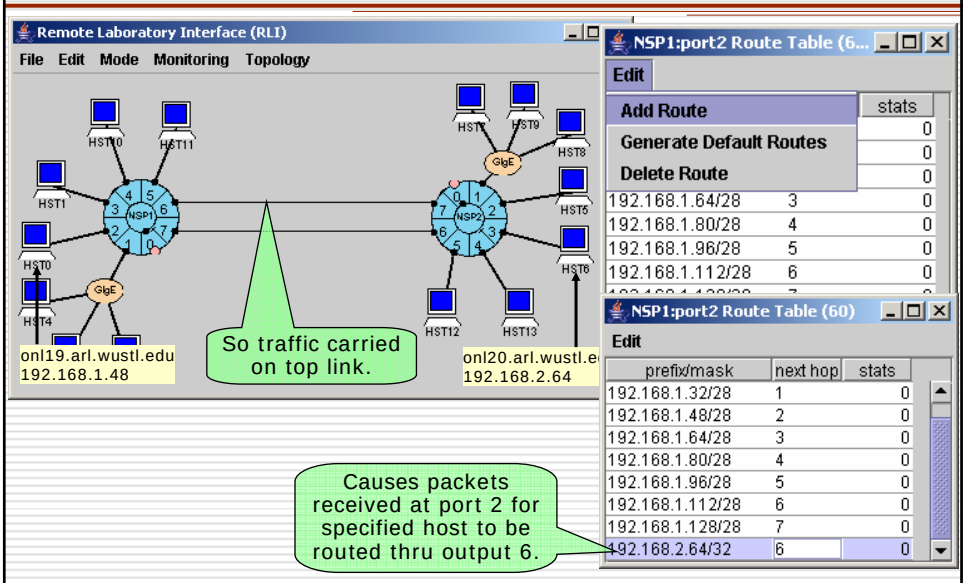
Configuring Routes



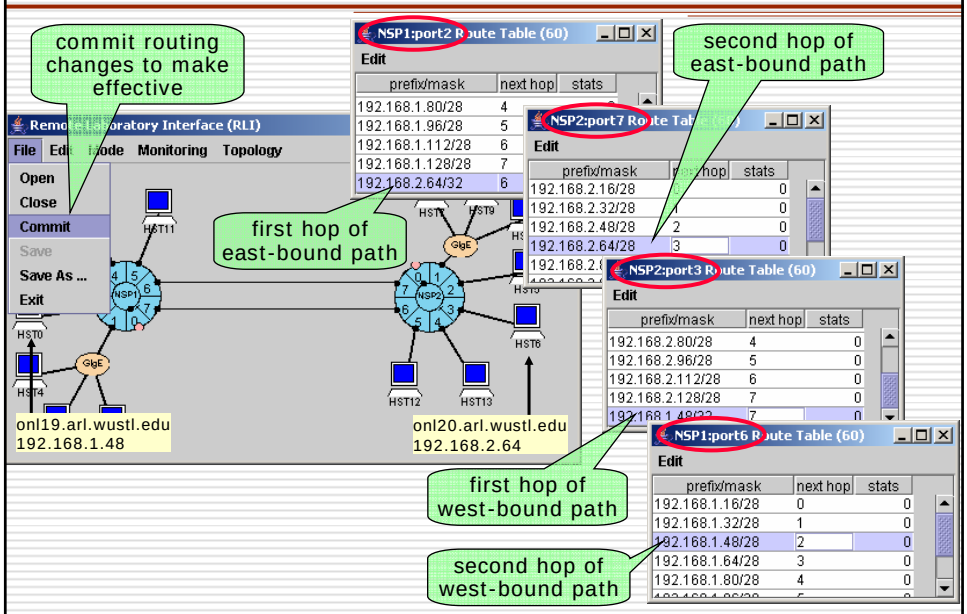
What does This Mean for Router?



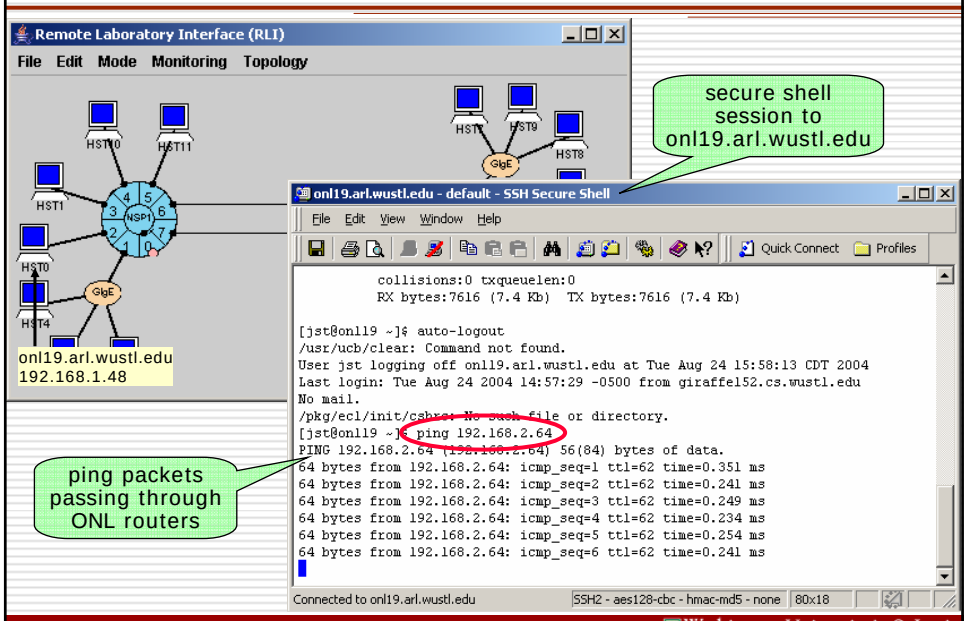
Adding More Routes



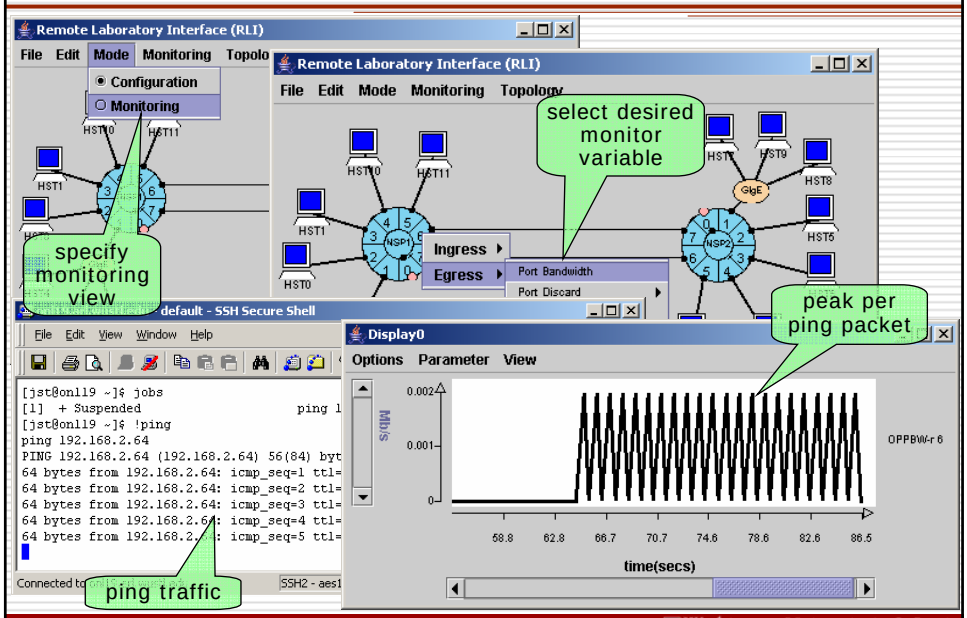
Routes for 2-Way Communication



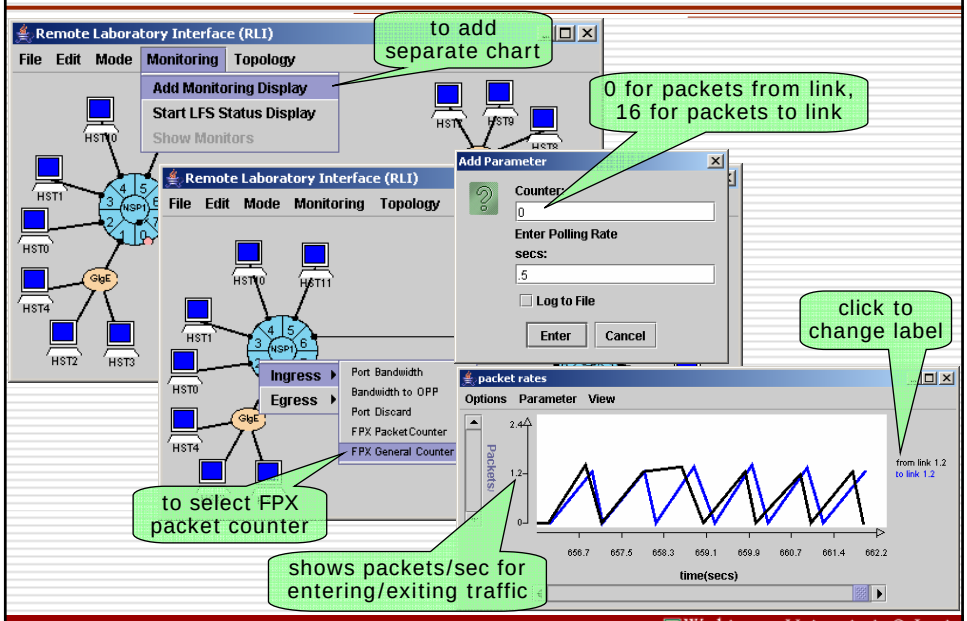
Verifying Routes



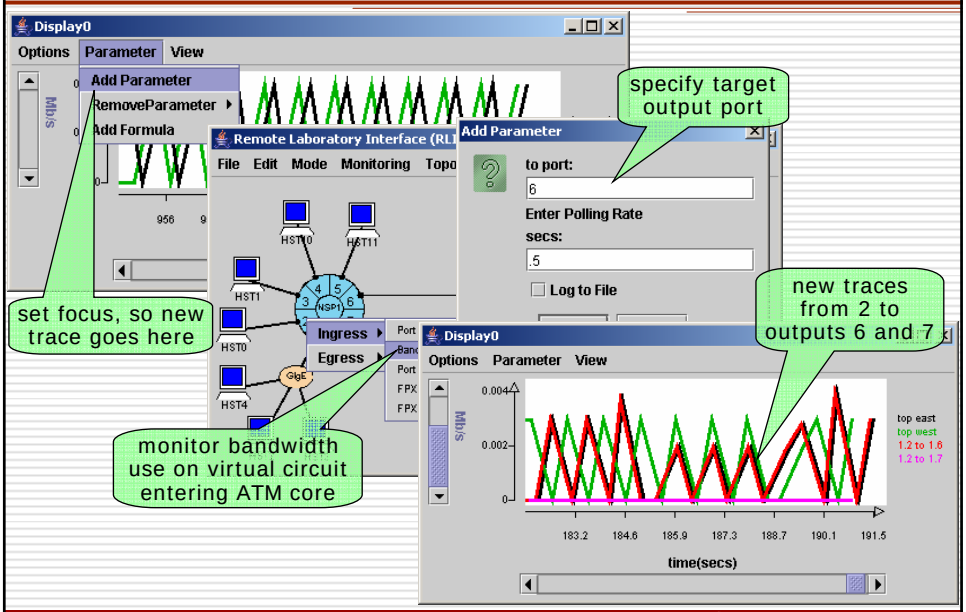
Monitoring Traffic



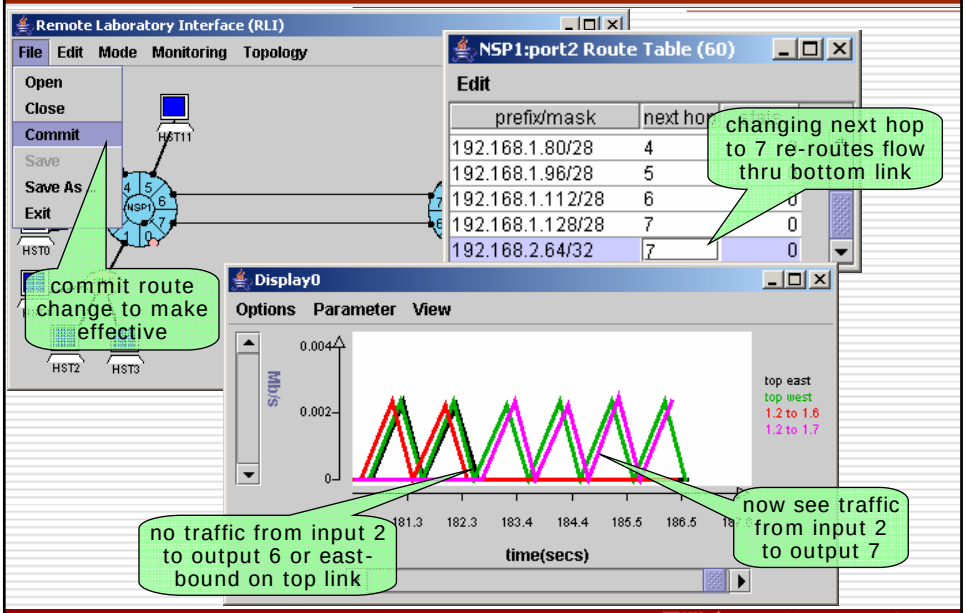
Monitoring Other Data



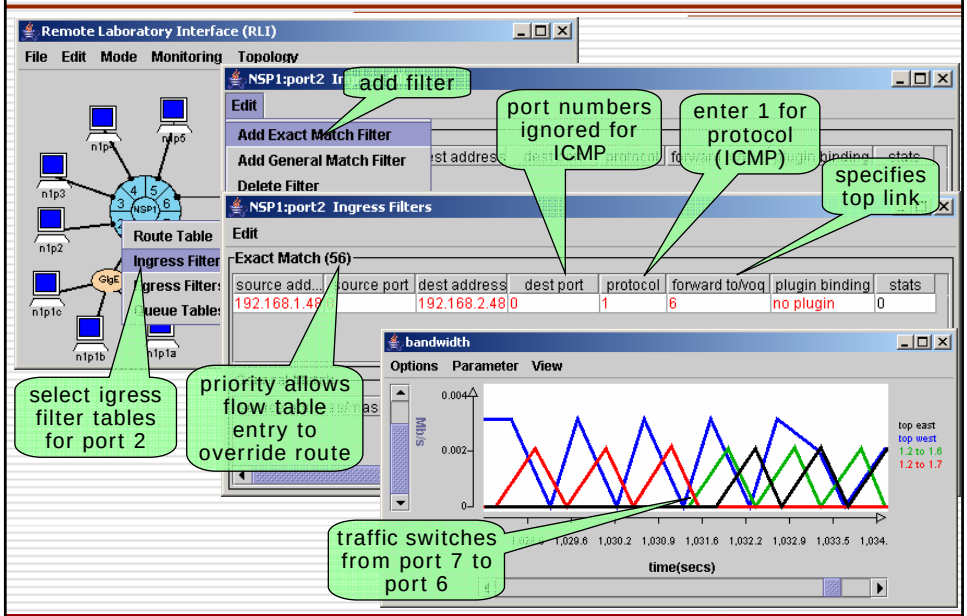
Monitoring Still Other Data



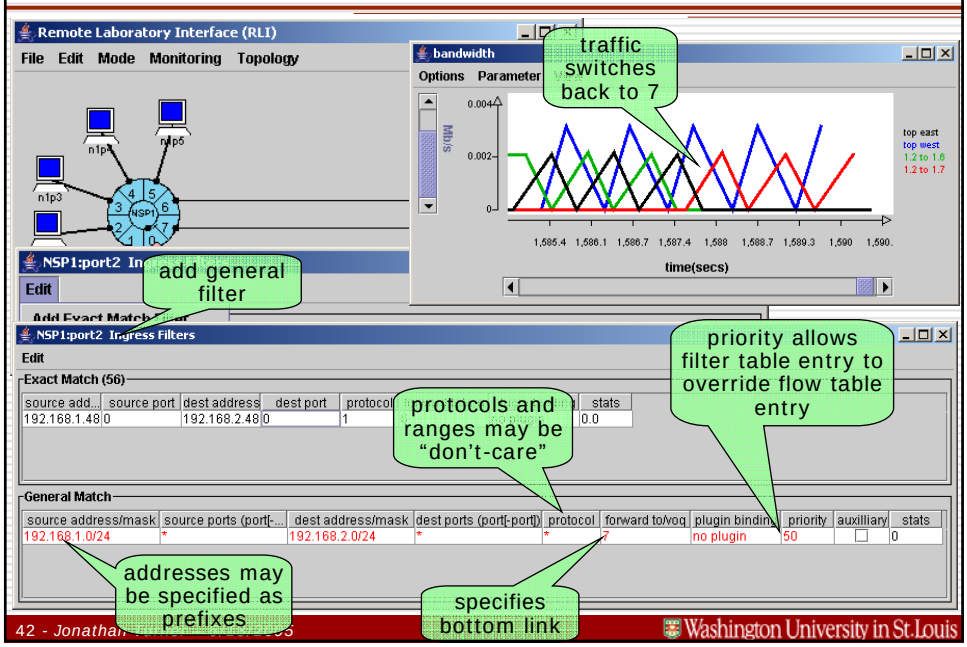
Changing Routes



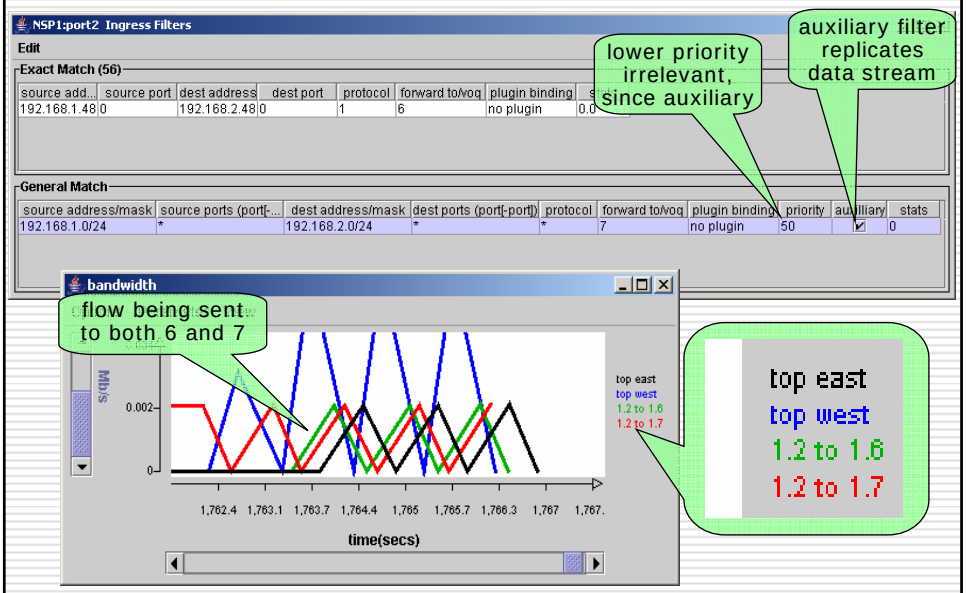
Using Flow Tables (Exact Match)



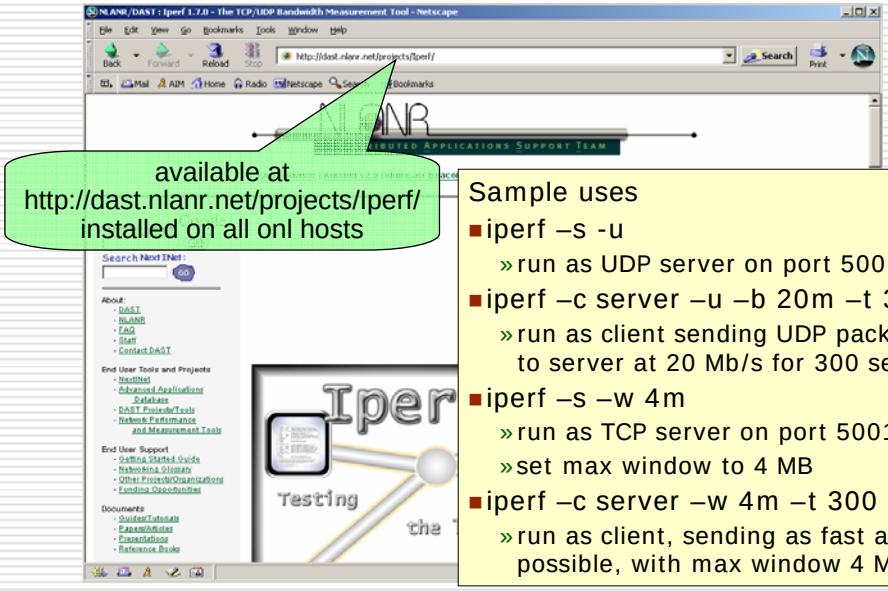
Using General Filter Tables



Using Auxiliary Filter to Copy Flow



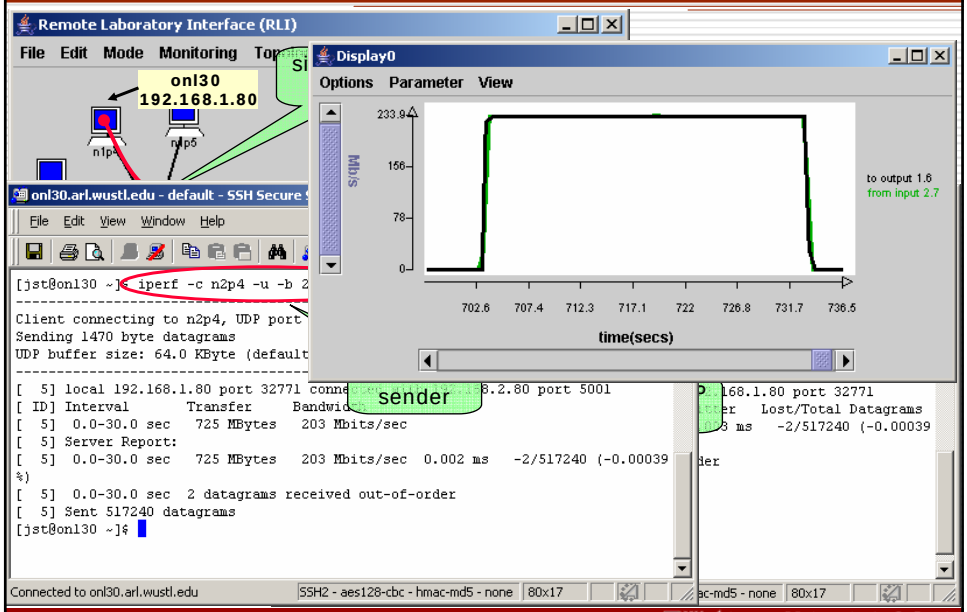
Generating Traffic with Iperf



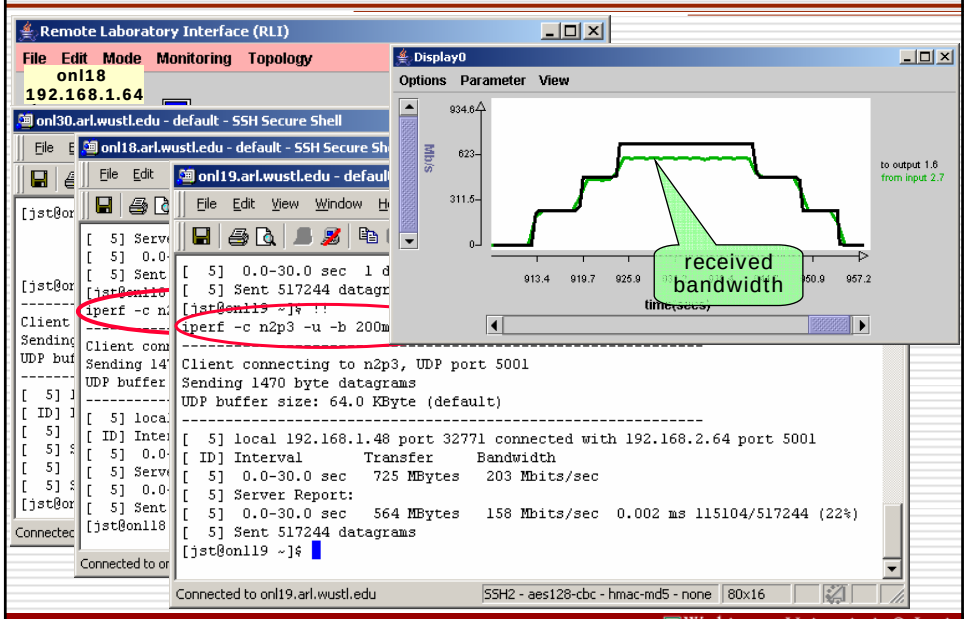
Sample uses

- 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

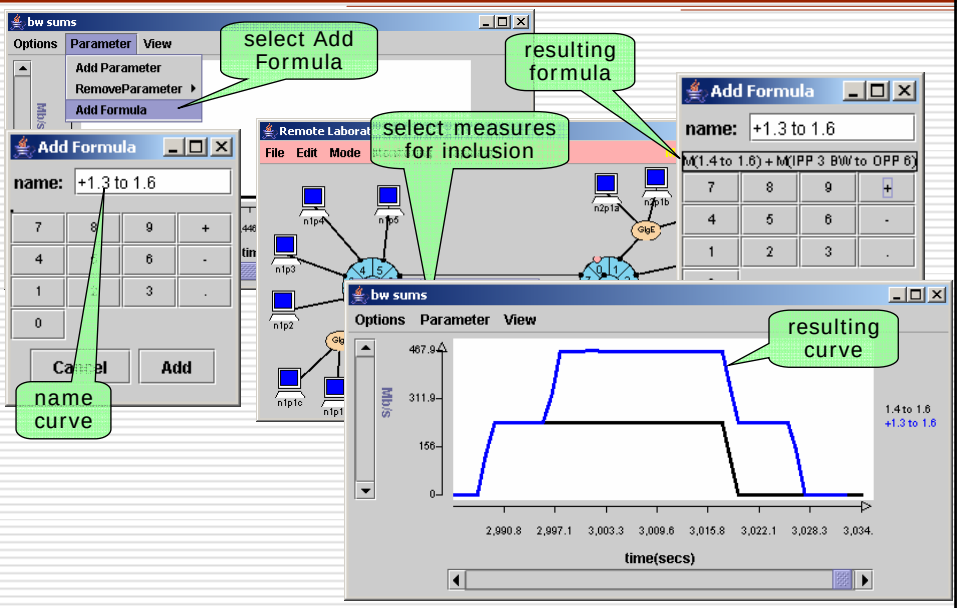
Using Iperf



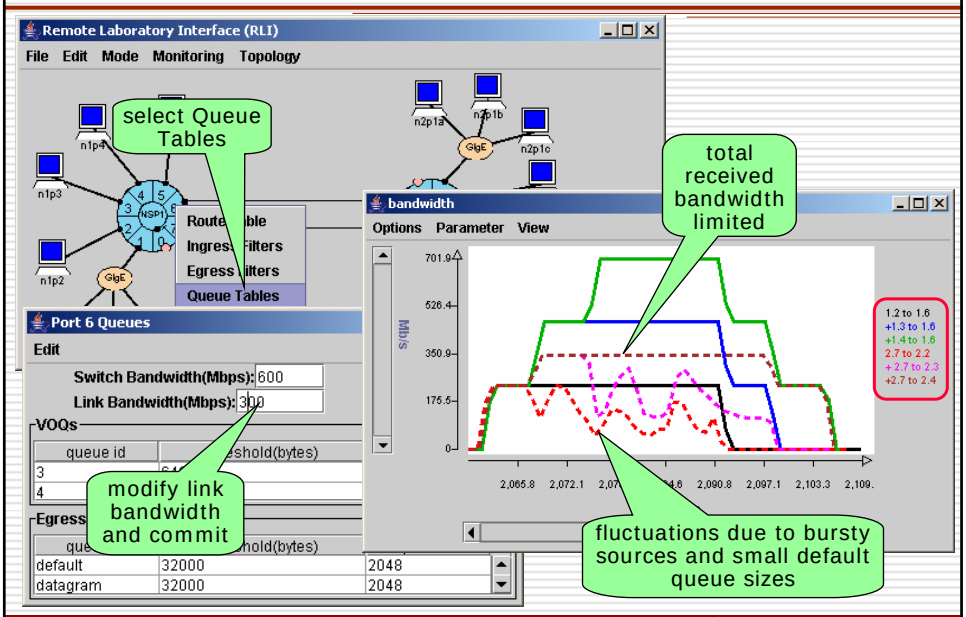
Multiple Iperf Streams



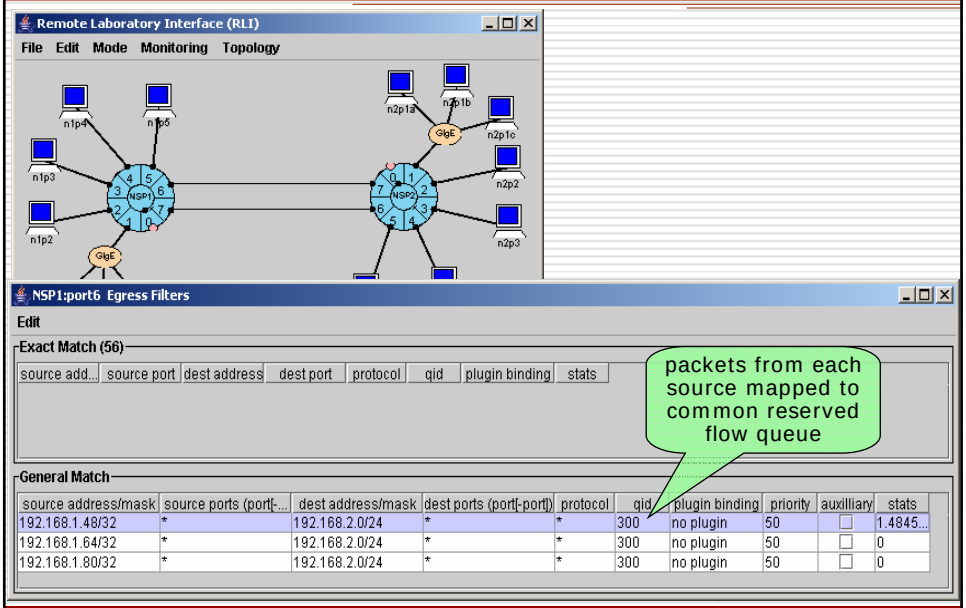
Displaying Incremental Bandwidth



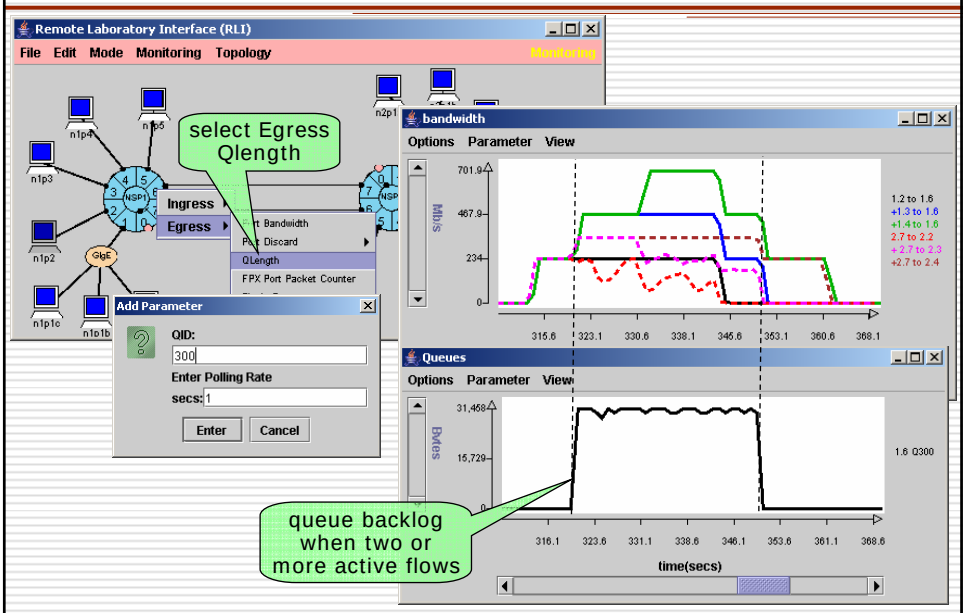
Modifying Link Rate



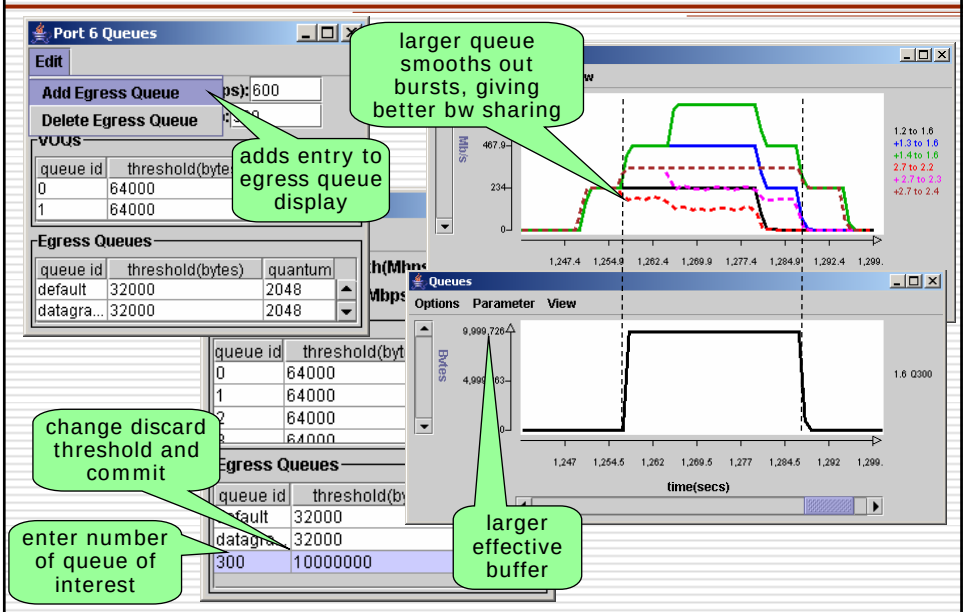
Mapping Flows to Single Queue



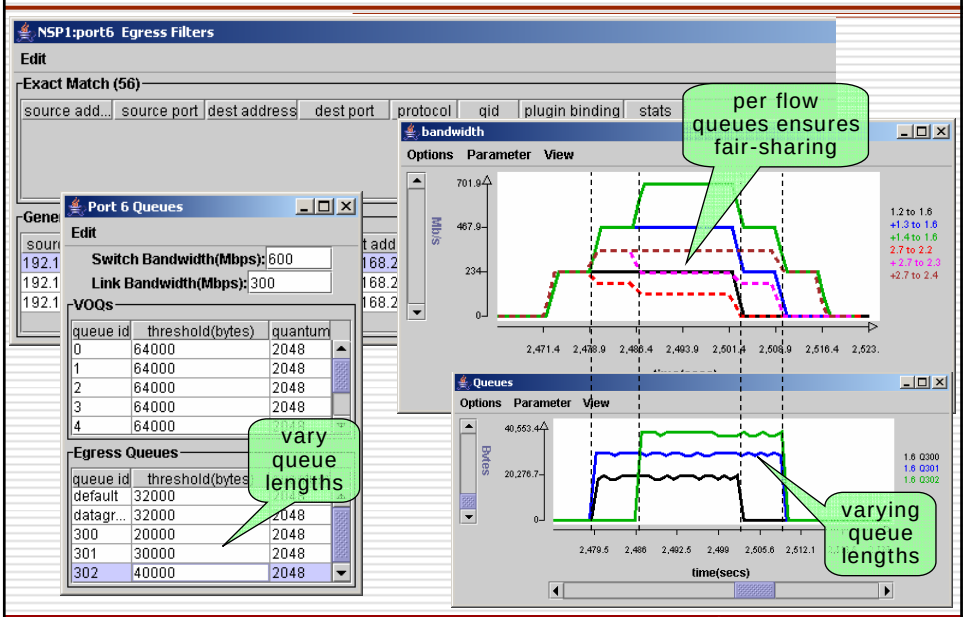
Monitoring Queue Length



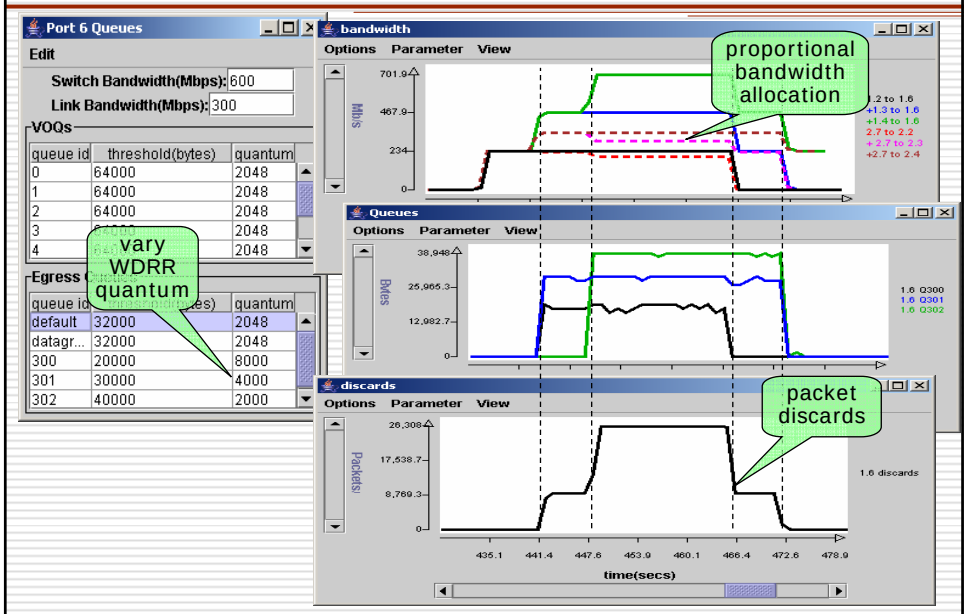
Changing Queue Size



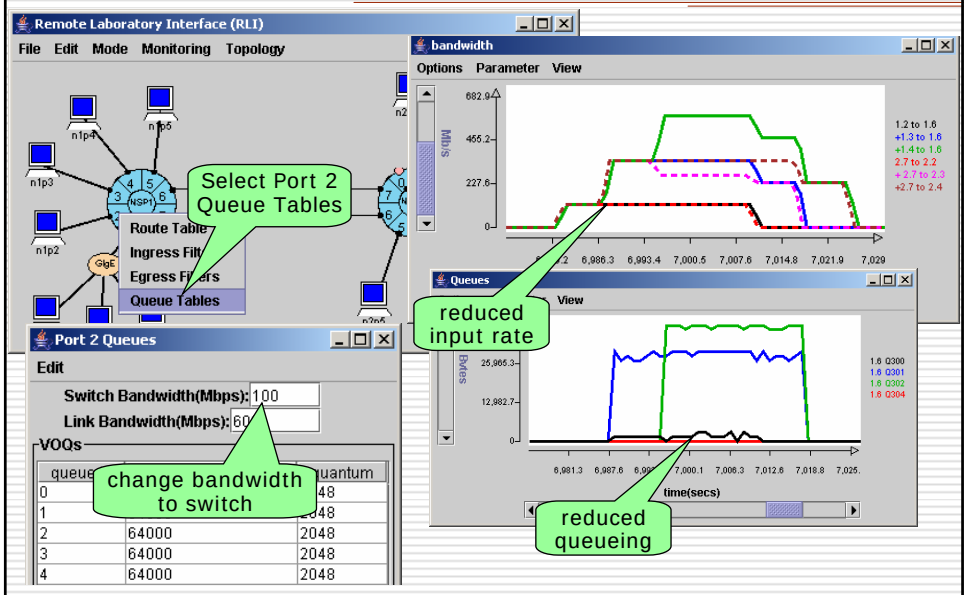
Mapping Flows to Separate Queues



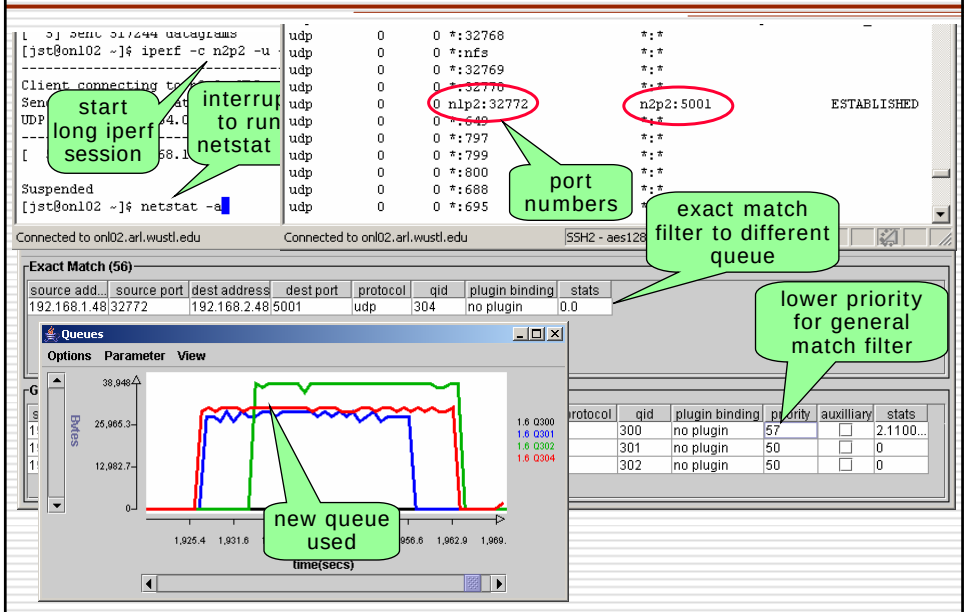
Changing Bandwidth Shares



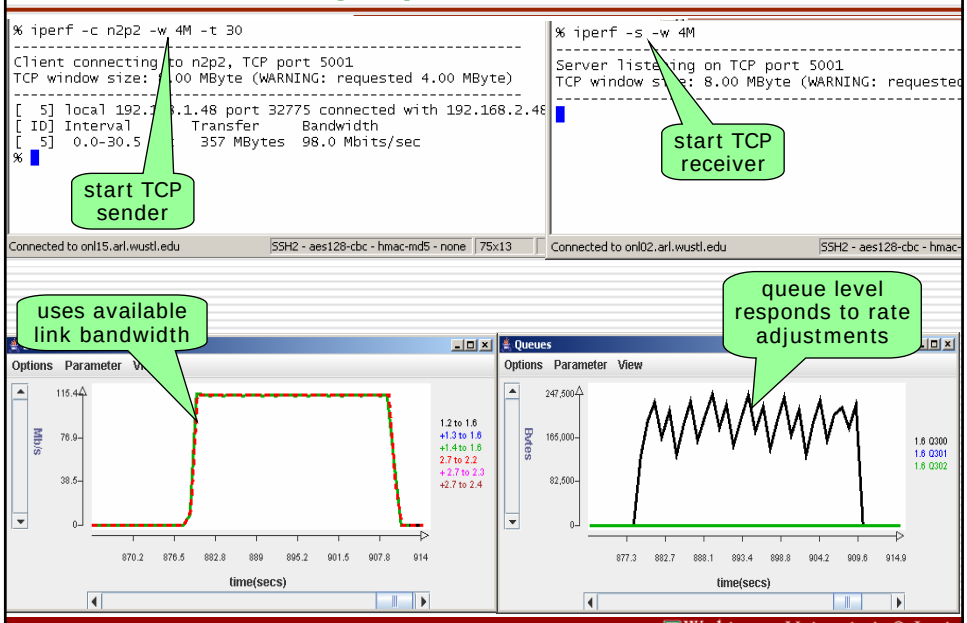
Changing VOQ Rates



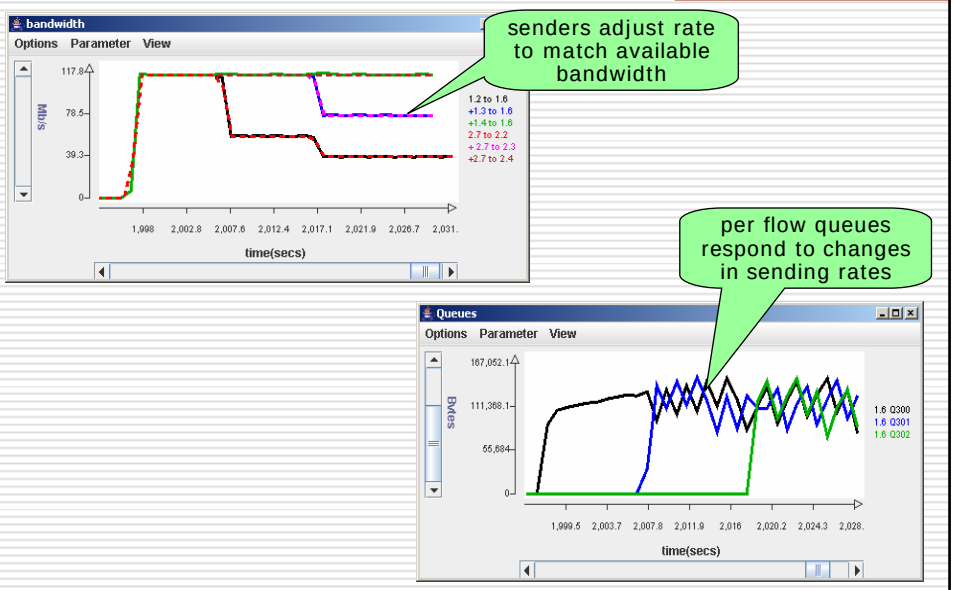
Using Exact Match Filters



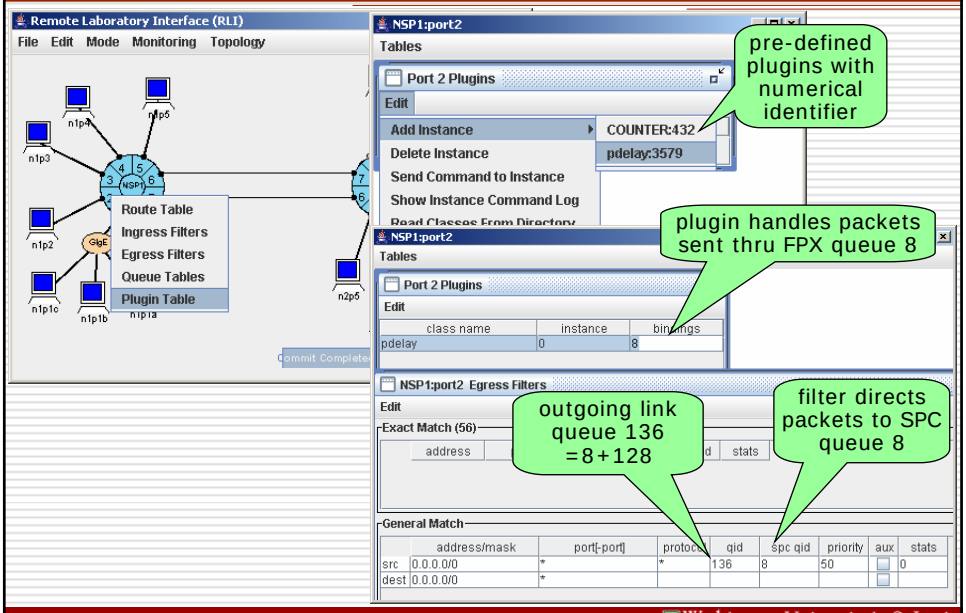
Using Iperf with TCP



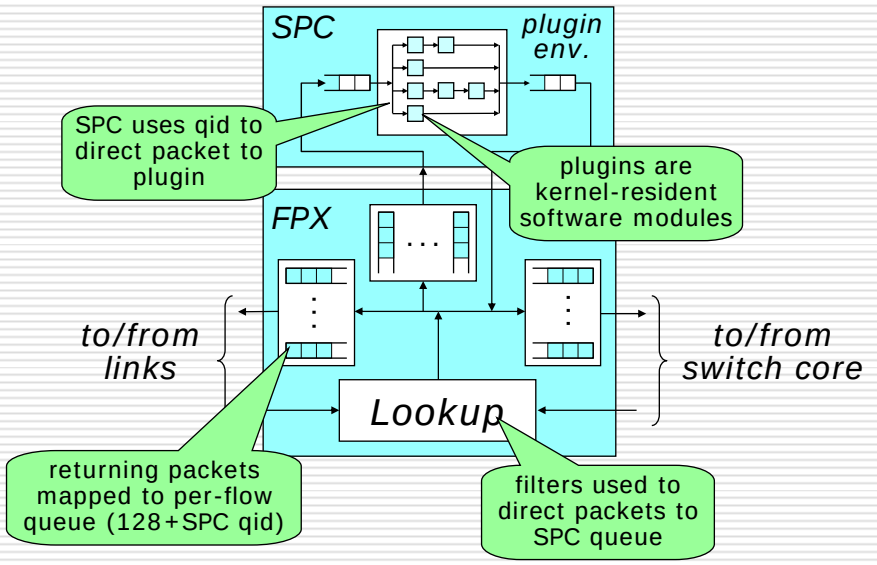
Competing TCP Flows



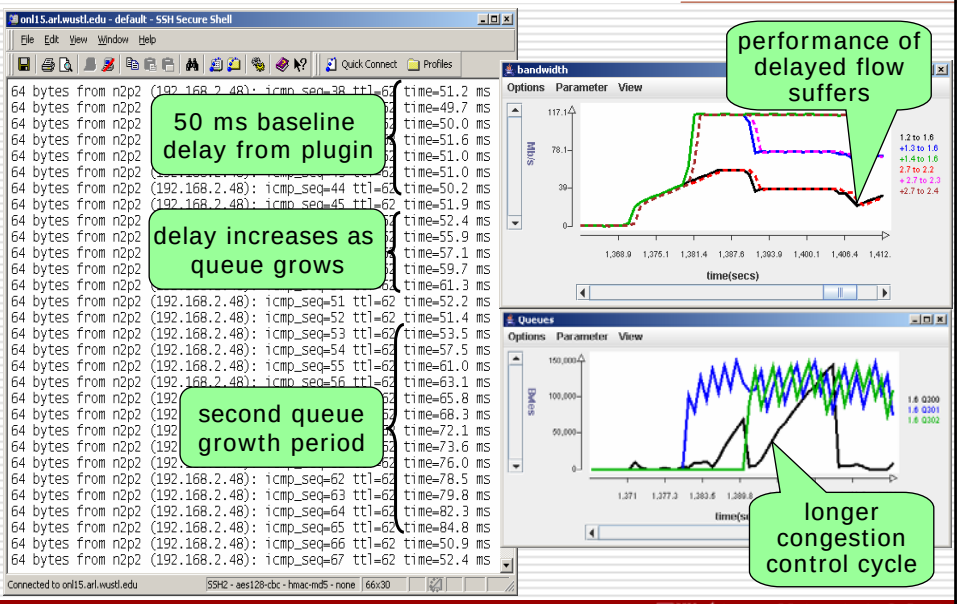
Adding SPC Plugins



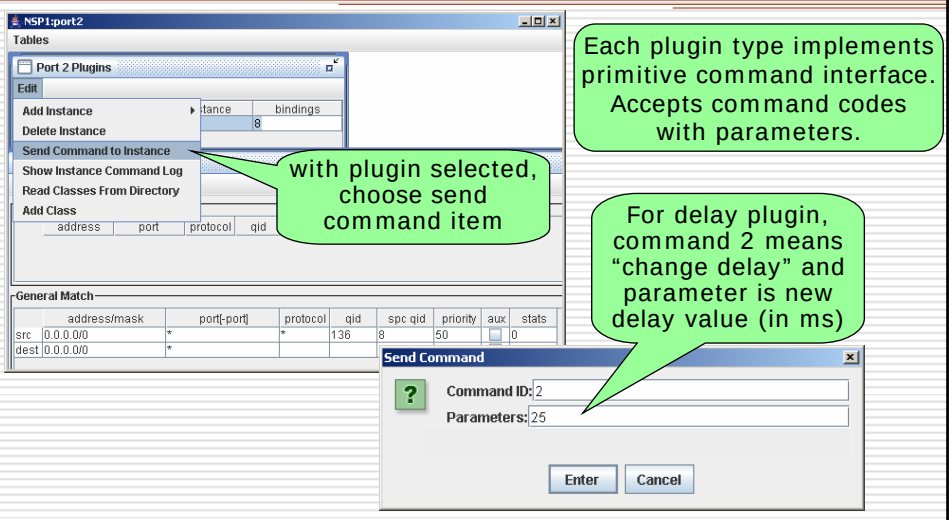
What Does it Mean?



Effect on TCP



Sending Messages to Plugins

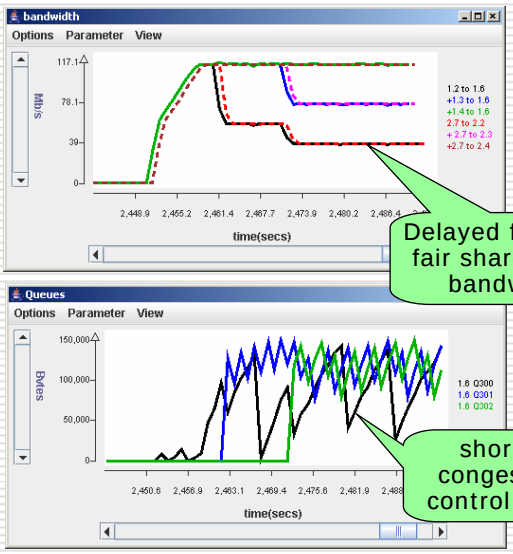


with plugin selected, choose send command item

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

```
#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);
```

numerical class id

per instance variables

standard function declarations change only names

myCounter_handle_packet

```
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

```
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) 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 = 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

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