The Open Network Lab
(Part 2)

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,
CNS-0551651, REL-0632580

Topics

1) Packet-pair example
   • meaning of “link bandwidth”
   • ~onl/.topology and ~onl/.topology.csh shell scripts

2) Iperf scripts

3) EM filters and GM filters
   • Flow tables and Filter tables

4) Mapping flows to reserved flow queues
   • a single reserved flow queue
   • separate reserved flow queues

5) Optional: Using iperf for TCP flows
Demos

1) Packet-pair example
   - link bandwidth (capacity)
   - ~/onl/topology and ~/onl/topology.csh shell scripts

2) UDP traffic demo with multiple flows
   - running iperf scripts from onlusr
   - link bandwidth, GM filters, queue size
   - single versus separate queues
   - with and without delay

---

The Packet-Pair Problem

- Apply Keshav’s idea
- If rcvr receives the two pkts at times t0 and t1,
  » What will be the value of t1-t0, the interpacket time? Why?
  » average interbit time?
- Describe an ONL experiment?
- Most general experiment?
Solving Problems

See
» www.arl.wustl.edu/~kenw/courses/cs422/resources/htmlfiles/how-to-solve-it.html

Steps
» Understand the problem
» Devise a plan
» Get unstuck
» Carry out the plan
» Reflect on the solution

• What are we given?
• What fundamental definitions and principles apply?
• What are the implications of the facts?
• Is there a clarifying figure, notation?

• Does solution make sense?
• Were there fundamental principles?
• Can we apply principles in a more general setting?

Sndr and Rcvr (~kenw/src/pkt-pair/)

sndr (client socket program)
» run on $n1p2
» sends 3 ping pkts to initialize ARP caches along path to n1p3
» sends ‘npkts’ UDP pkts to ‘rcvr’ interface using sendto(2)
  • inserts sender timestamp (gettimeofday(3)) right before sending
» reads ‘npkts’ ACK pkts
» compute statistics
» UDP packet format
  • 1470-byte payload has application header
  > 8-byte seqnum string, int sequence number, 4 timestamps
  • timestamp is value returned from gettimeofday(3)

rcvr (server socket program)
» run on $n1p3
» for each UDP pkt
  • read pkt into buffer
  • insert timestamp (gettimeofday(3)) into application header
  • send ACK pkt to sender

1498-byte datagram (includes bytes in header)
Packet-Pair Experiment (1)

Puts $n1p2, etc. into Linux csh environment

20-byte IP hdr
8-byte UDP hdr
1470-byte payload

rcvr: n1p3
#pkts: 7
#runs: 2

Run 1

RTTs (msec):
0.48 0.39 1.30 1.70 4.10 5.78 7.15
Srcr Interop Times (usec):
37 913 1488 1494 1519 1477
Bandwidth est. at sndc [Mbps]:
139.00 148.00 331.09 992.27 1997.33 400.92
Bandwidth est. at rcvr [Mbps]:
333.09 15.19 8.05 8.02 7.69 6.31

Run 2

RTTs (msec):
1.41 2.80 4.29 5.78 7.27 8.80 10.30
Srcr Interop Times (usec):
33 7 6 5 16 9
Bandwidth est. at sndc [Mbps]:
1497 1497 1497 1498 1540 1467
Bandwidth est. at rcvr [Mbps]:
1434.15 1712.30 1997.33 2356.80 756.20 1714.00

Packet-Pair Experiment (2)

14 pkts x ~12000 bits/pkt / 0.25 sec = 0.672 Mbps

ARP pkts

View ➔ Show Values
Queue Table ➔ Link Bandwidth

- Controlled by a token bucket model
  - \( r \): long-term average link rate (RLI parameter)
    - 54 Kbps granularity
  - \( b \): maximum bucket depth (4000 bytes)
  - \( R \): physical link rate (2 Gbps)

- Effect
  - avg output rate is \( r \)
  - peak rate is \( R \)

- Operational definition
  - fill token bucket at rate \( r \)
  - transmit pkt when \#tokens ≥ length of pkt at head of queue

- *link regulator* model, NOT link emulator

Files in ~kenw/src/pkt-pair/

- README
  - documentation: describes algorithm
- stdinc.h
  - wrappers for socket, bind, sendto, rcvfrom, etc.
- pkt-pair.h
  - application header definition “struct pkthdr {...}”
  - constants and tvdiff() function
- pp-sndr.c
  - UDP sender (client program)
- pp-rcvr.c
  - UDP receiver (server program)
- Makefile
  - ‘make’ will compile sender and receiver
Do analysis before coding
  » How long to transmit 500 1000-byte pkts?
    • if no pkt losses?
    • when will bottleneck queue overflow?

Can I use pkt-pair code?
  » Read code ➔ general behavior?
    • NOTE: will NOT handle pkt drops or ACK drops
  » Modifications required to do simple test?
    • My suspicion
      ➔ need to only change a few lines
      ➔ may encounter ACK drops at sender because of UDP buffers
        ➔ but should work for 200 or 300 pkts
    • If ACK drops occur at sndr, sndr needs to read ACKs before overflow occurs ➔
      ➔ Minimally reorganize doClnt() function

Can I use iperf to verify results?
Some Bottleneck Experiments

**Configuration**

- Senders: n1p2, n1p3
- Receivers: n1p4, n1p5
- **Bottleneck** link 1.7-1.6 (300 Mbps)
- Queue 300: 150,000 bytes
- Forward path through 1.7-1.6
- Return path through 1.6-1.7

**Flows**

- 8 second staggered starting time (shell script)
- Use *iperf* traffic generator

Two 200 Mbps UDP flows through bottleneck

- Pkt scheduling: Equal rate vs. 1:3 rate

Two TCP flows through bottleneck

- With and without 50 msec ACK delay

---

**Iperf Server And Client Scripts**

```bash
#!/bin/sh
source /users/tml/topology # get def of external interfaces

# start udp receivers
for host in $n1p4 $n1p5; do
    ssh $host /usr/local/bin/iperf -s -u &
done

# start tcp sender
while true; do
    ssh $n1p2 /usr/local/bin/iperf -c $n1p4 -t 20m -b 200m -a
    sleep 1
    ssh $n1p3 /usr/local/bin/iperf -c $n1p5 -t 20m -b 200m -a
    sleep 50
done
```

- Puts $n1p2, etc. into Linux bash environment
- Run "iperf -s -u" on hosts $n1p4 and $n1p5
- Send at 200 Mbps for 20 seconds
- Run as UDP server
- Run as UDP client sending to n1p4
Running Receiver and Sender Scripts

Send rate: 10 Gbps | Receiver rate: 10 Gbps

1) Run receiver script
2) Run sender script

Demo 2a (iperf scripts run from onlusr)

- ~onl/.topology and ~onl/.topology.csh
  - $n1p1a, $n1p1b, $n1p1c, $n1p2, $n1p3, etc.
- pkill -n iperf
- ~onl/export/Examples/\{urcvrs,usndrs\}-1nsp
Demo 2 (Two UDP Flows)

- Monitoring
  - Bandwidth, Queue length, Pkt drops
- One queue (300) for all flows
- Separate queues (300, 301) for each flow
  - Sharing bandwidth (weighted Deficit Round Robin)

Configuring For Demo 2

- At egress port 7 (the bottleneck)
  - Install GM filter to direct traffic to queue 300
  - Configure queue 300 to be 150,000 bytes
- Monitor VOQ VCI bw, queue 300 length, and pkt drops
Packet Processing With GM Filter

Port 2 (In)

Virtual Output Queues
voq 0
voq 7

ATM switch fabric

Port 7 (Out)

dgram

qid 300
res flow

Port 2 (Ingress Side)

Port 7 (Egress Side)

Monitoring Points

Port 2
(Ingress Side)
(qids 504-511)

Port 6
(Ingress Side)

Port 4
(Egress Side)

Port 6
(Egress Side)

Port 7
(Ingress Side)

ATM Switch

Port 7 Link Capacity and Queue 300

Add GM Filter and Configure Queue 300

- **GM filter**
  - matches all pkts
  - protocol (*) matches any protocol
  - Queue 300
  - Priority 50 (higher than RTs 60)

- **Queue 300**
  - 150,000 bytes

- **Egress link capacity = 300 Mbps**
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 – general match
    - (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 (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.
**VOQ Bandwidth and Q300 Length**

- Both senders send at 200 Mbps
  - 1.2 to 1.7 and 1.3 to 1.7
- Only n1p2 traffic
  - 200 Mbps goes to n1p4
- n1p2 and n1p3 traffic
  - 120 Mbps to n1p4 (1.6-1.4)
  - 180 Mbps to n1p5 (1.6-1.5)
- Queue 300 is full
- Only n1p3 traffic
  - 200 Mbps goes to n1p5

---

**Monitoring Port 7 Egress Pkt Drops**

- See Tutorial => Summary Information => FPX Counters
- Click label
Separate Queues For Each Flow

150 Mbps each flow
Q300 and Q301 filled

n1p3 Share = 3X n1p2 Share

very little queuing at queue 300
Delay ACK Pkts By 50 Msec

- Define GM filter at egress port 6
  - matches all pkts
  - directs traffic to SPC delay plugin
- Define standard delay plugin
  - default delay is 50 msec
  - change delay thru msg interface
- Binding between filter and plugin
  - SPC QID (a value between 8 and 127)

Delay ACK Pkts By 20 Msec

#pkts  #drops  #fwd  max qlen  delay  callback
End Demo 2

TCP Scripts

4 MB receiver window to handle bandwidth-delay product