Problem Set 2 Solutions

1. Trace files [2 points]. *Create a router configuration file called* `prob1.click` *that reads packets from the* `tcpdump` *trace file '*`f1a.dump`*' and writes those packets unchanged to the* `tcpdump` *trace file '*`f1b.dump`*'. The Click driver should exit when* `f1a.dump` *is out of packets.*

```
FromDump(f1a.dump, STOP true) -> ToDump(f1b.dump, ENCAP IP)
```

2. Routing [2 points]. *Create a router configuration file called* `prob2.click` *that reads packets from the* `tcpdump` *trace file '*`f2a.dump`*' and routes those packets using the following routing table:*

Destination prefix	***Output trace file***
131.0.0.0/8	`f2b.dump`
131.179.0.0/16	`f2c.dump`
18.0.0.0/8	`f2d.dump`
All others	`f2e.dump`

IP routing elements like *StaticIPLookup* and *LinearIPLookup* were the obvious choices.

```
FromDump(f2a.dump, STOP true)
   -> r :: StaticIPLookup(131.0.0.0/8 0, 131.179.0.0/16 1, 18.0.0.0/8 2, 0/0 3)
r[0] -> ToDump(f2b.dump, ENCAP IP)
r[1] -> ToDump(f2c.dump, ENCAP IP)
r[2] -> ToDump(f2d.dump, ENCAP IP)
r[3] -> ToDump(f2e.dump, ENCAP IP)
```

But there were other choices too – in particular, *IPClassifier* and *IPFilter*.

```
FromDump(f2a.dump, STOP true)
   -> ip :: IPClassifier(dst 131.179.0.0/16, dst 131.0.0.0/8, dst 18.0.0.0/8, -)
      // 131.179.0.0/16 must come first because filters are checked in order
ip[0] -> ToDump(f2c.dump, ENCAP IP)
ip[1] -> ToDump(f2b.dump, ENCAP IP)
ip[2] -> ToDump(f2d.dump, ENCAP IP)
ip[3] -> ToDump(f2e.dump, ENCAP IP)
```

A common *IPClassifier* error was to forget the '`dst`' modifiers. If you leave off '`dst`', *IPClassifier* looks at *both* source and destination addresses, so all packets with source address 131.179.0.1 would get sent to `f2c.dump` no matter where they were headed.

3. Error checking [3 points]. *Create a router configuration file called* `prob3.click`*, based on* `prob2.click`*. The configuration should read from '*`f3a.dump`*' and write to '*`f3*.dump`*'. But this time, you should check for the following six kinds of errors, in this order.*

Problem	***Action***
Invalid IP header or checksum	*Discard packet*
Invalid TCP header or checksum	*Discard packet*
Invalid UDP header or checksum	*Discard packet*
Invalid ICMP header or checksum	*Discard packet*
Expired IP TTL	*Generate appropriate ICMP error message, send message to '*`f3f.dump`*'*
Packet longer than 1500 bytes	*Discard packet*

```
FromDump(f3a.dump, STOP true)
   -> CheckIPHeader
   -> i1 :: IPClassifier(tcp, udp, icmp, -)
   -> CheckTCPHeader
   -> ttl :: IPClassifier(ttl > 0, -)
   -> cl :: CheckLength(1500)
   -> ip :: IPClassifier(dst 131.179.0.0/16, dst 131.0.0.0/8, dst 18.0.0.0/8, -)

i1[1] -> CheckUDPHeader -> ttl
i1[2] -> CheckICMPHeader -> ttl
i1[3] -> ttl

ttl[1] -> ICMPError(18.26.7.1, timeexceeded, transit)
   -> ToDump(f3f.dump, ENCAP IP)

ip[0] -> ToDump(f3c.dump, ENCAP IP)
ip[1] -> ToDump(f3b.dump, ENCAP IP)
ip[2] -> ToDump(f3d.dump, ENCAP IP)
ip[3] -> ToDump(f3e.dump, ENCAP IP)
```

Common errors included dropping all non-TCP/UDP/ICMP packets and not checking for the errors in order.

***4. Handlers** [**2 points**]. Create a router configuration file called `prob4.click`, based on `prob3.click`. It should read from '`f4a.dump`' and write to '`f4*.dump`'. But this time, in addition to checking for errors, running the configuration should generate a file '`f4.drops`' containing the following 6 lines:*

1. *The number of packets with invalid IP headers or checksums;*
2. *The number of packets with invalid TCP headers or checksums;*
3. *The number of packets with invalid UDP headers or checksums;*
4. *The number of packets with invalid ICMP headers or checksums;*
5. *The number of packets with expired IP TTLs; and*
6. *The number of packets longer than 1500 bytes.*

Each packet will show up in at most one of these counts.

```
FromDump(f4a.dump, STOP true)
   -> cip :: CheckIPHeader
   -> i1 :: IPClassifier(tcp, udp, icmp, -)
   -> ctcp :: CheckTCPHeader
   -> ttl :: IPClassifier(ttl 0, -) [1]
   -> cl :: CheckLength(1500)
   -> ip :: IPClassifier(dst 131.179.0.0/16, dst 131.0.0.0/8, dst 18.0.0.0/8, -)

i1[1] -> cudp :: CheckUDPHeader -> ttl
i1[2] -> cicmp :: CheckICMPHeader -> ttl
i1[3] -> ttl

ttl[0] -> cttl :: Counter
   -> ICMPError(18.26.7.3, timeexceeded, transit)
   -> ToDump(f4f.dump, ENCAP IP)

ip[0] -> ToDump(f4c.dump, ENCAP IP)
ip[1] -> ToDump(f4b.dump, ENCAP IP)
ip[2] -> ToDump(f4d.dump, ENCAP IP)
ip[3] -> ToDump(f4e.dump, ENCAP IP)
```

```
cl[1] -> ccl :: Counter -> Discard

DriverManager(wait_stop, save cip.drops f4.drops,
   append ctcp.drops f4.drops,
   append cudp.drops f4.drops,
   append cicmp.drops f4.drops,
   append cttl.count f4.drops,
   append ccl.count f4.drops)
```

5. Compound elements [3 points]. *Create a partial router configuration file called `prob5.click`. This file should define a compound element named '`ErrorChecker`' with one input and one output that implements all the error checks from Problem 3. All erroneous packets should be dropped (not written to a file).*

```
elementclass ErrorChecker {
   input -> cip :: CheckIPHeader
      -> i1 :: IPClassifier(tcp, udp, icmp, -)
      -> ctcp :: CheckTCPHeader
      -> ttl :: IPFilter(drop ttl 0, allow all)
      -> cl :: CheckLength(1500)
      -> output;
   i1[1] -> cudp :: CheckUDPHeader -> ttl
   i1[2] -> cicmp :: CheckICMPHeader -> ttl
   i1[3] -> ttl
}
```

A couple people had syntax errors! It's always worthwhile to check your work, even if the instructor hasn't provided a handy '`make check`' command.

I intended for you to create an `elementclass` definition, but the problem was poorly worded, so element declarations were also accepted. Another poor wording issue: It wasn't clear to everyone whether bad-TTL packets should be simply dropped (as in the solution above), or whether you should generate ICMP errors and emit them out the first output. You could probably guess the intention by looking at Problem 6, but either was acceptable.

6. Compound element overloading [3 points]. *Create a partial router configuration file called `prob6.click`, based on `prob5.click`. This time, the `ErrorChecker` compound element should support three different use patterns. If it is used with one input and one output, it should behave like in Problem 5. If it is used with one input and two outputs, then all erroneous packets should be emitted on the second output (not dropped). If it is used with one input and seven outputs, then the first output is for correct packets, and the following six outputs are used for the six different errors.*

The most "readable" solutions used an "element superclass", like so.

```
elementclass ErrorCheckerImpl {
   input -> cip :: CheckIPHeader
      -> i1 :: IPClassifier(tcp, udp, icmp, -)
      -> ctcp :: CheckTCPHeader
      -> ttl :: IPFilter(1 ttl 0, 0 all)
      -> cl :: CheckLength(1500)
      -> output;
   i1[1] -> cudp :: CheckUDPHeader -> ttl
   i1[2] -> cicmp :: CheckICMPHeader -> ttl
   i1[3] -> ttl
   cip[1] -> [1]output
```

```
   ctcp[1] -> [2]output
   cudp[1] -> [3]output
   cicmp[1] -> [4]output
   ttl[1] -> [5]output
   cl[1] -> [6]output
}

elementclass ErrorChecker {
   input -> e :: ErrorCheckerImpl -> output
   e[1] -> d :: Discard
   e[2] -> d; e[3] -> d; e[4] -> d; e[5] -> d; e[6] -> d
||
   input -> e :: ErrorCheckerImpl -> output
   e[1] -> [1]output; e[2] -> [1]output; e[3] -> [1]output
   e[4] -> [1]output; e[5] -> [1]output; e[6] -> [1]output
||
   input -> e :: ErrorCheckerImpl -> output
   e[1] -> [1]output; e[2] -> [2]output; e[3] -> [3]output
   e[4] -> [4]output; e[5] -> [5]output; e[6] -> [6]output
}
```

Interestingly, you can avoid the separate *ErrorCheckerImpl* class with an ellipsis:

```
elementclass ErrorChecker {
   // first, define the version with 7 outputs
   input -> cip :: CheckIPHeader
      -> i1 :: IPClassifier(tcp, udp, icmp, -)
      -> ctcp :: CheckTCPHeader
      -> ttl :: IPFilter(1 ttl 0, 0 all)
      -> cl :: CheckLength(1500)
      -> output
   i1[1] -> cudp :: CheckUDPHeader -> ttl
   i1[2] -> cicmp :: CheckICMPHeader -> ttl
   i1[3] -> ttl
   cip[1] -> [1]output
   ctcp[1] -> [2]output
   cudp[1] -> [3]output
   cicmp[1] -> [4]output
   ttl[1] -> [5]output
   cl[1] -> [6]output
}

elementclass ErrorChecker {
   // overload the 7-output version using its code
   input -> e :: ErrorChecker -> output
   e[1] -> d :: Discard; e[2] -> d; e[3] -> d; e[4] -> d; e[5] -> d; e[6] -> d
||
   input -> e :: ErrorChecker -> output
   e[1] -> [1]output; e[2] -> [1]output; e[3] -> [1]output
   e[4] -> [1]output; e[5] -> [1]output; e[6] -> [1]output
||
   ...
}
```

***7. Scheduling** [**5 points**]. Create a router configuration file called `prob7.click`. It should read packets from '`f7a.dump`', maintaining the timing of the dump file. (This means that if two adjacent packets in the*

dump file have timestamps that are 0.5 seconds apart, the packets will be emitted into the configuration 0.5 seconds apart.) The packets should be written into '`f7b.dump`', except that:

- *Packets should be written to the dump at a maximum rate of 384 Kb/s. Since the input rate might be more than 384 Kb/s, this means you will need queuing and traffic shaping in your configuration.*
- *TCP should always get up to 75% of the bandwidth (up to 288 Kb/s). This means that if the TCP input rate is 288 Kb/s and the UDP input rate is also 288 Kb/s, TCP's output rate will equal 288 Kb/s and UDP's output rate will be limited to 96 Kb/s. If the TCP input rate is less than 288 Kb/s, UDP and other kinds of traffic can take up the remainder.*

Packet timestamps in the output file should reflect the 384 Kb/s rate limit.

This was the killer problem. There were three common errors: assuming uniform packet sizes, separating rates *before* the *Queue*s, and allowing reordering among packets in the same stream.

The problem asked you to limit traffic streams to given numbers of bits per second. Obviously, you need to know how many bits each packet contains. Some solution attempts used *RoundRobinSched* or *StrideSched* to schedule between different queues – basically, three pulls from the "TCP queue" for every pull from the "other queue". But *RoundRobinSched* and *StrideSched* schedule *per packet*, indifferent to the number of bits per packet. If all the UDP packets were huge and all the TCP packets were small, a 3:1 packet ratio might correspond to a 1:12.5 byte ratio. You need to shape the traffic to a 3:1 *byte* ratio.

Many people tried to use *BandwidthRatedSplitter*s on the *input* side, like this:

```
FromDump(f7a.dump, STOP true, TIMING true)
   -> ipc :: IPClassifier(tcp, -);

ipc[0] -> tcp_split :: BandwidthRatedSplitter(36000);
  tcp_split[0] -> q_tcp :: Queue;
  tcp_split[1] -> q_overflow :: Queue;
ipc[1] -> udp_split :: BandwidthRatedSplitter(12000);
  udp_split[0] -> q_udp :: Queue;
  udp_split[1] -> q_overflow;

p :: PrioSched;
q_tcp -> [0] p;  q_udp -> [1] p;  q_overflow -> [2] p;
p -> BandwidthRatedUnqueue(48000) -> SetTimestamp -> ToDump(f7b.dump, ENCAP IP)
```

This approach has two problems, one serious and one not so serious. Not so serious first: This approach can reorder packets from the same flow. For TCP, for example, packet 1 may get sent to `q_tcp`, packet 2 to `q_overflow`, and packet 3 to `q_tcp`; then the packets might get emitted in the order 1, 3, 2. This kind of reordering can kill TCP performance; it's best to avoid it when possible.

The more serious problem is that the early *BandwidthRatedSplitter*s didn't seem to work! No one using a *BandwidthRatedSplitter* got their approach to `make check`. (There were either too few TCP packets or too few non-TCP packets.) I don't have a good intuition for the reason. A Click bug is one possibility, but the code looks good. More likely, there's an inherent timing problem in limiting bandwidth *before* the queueing stage: some bad interaction between the upstream limits and the downstream, 48 Kb/s limit.

As usual, the simplest solution is best.

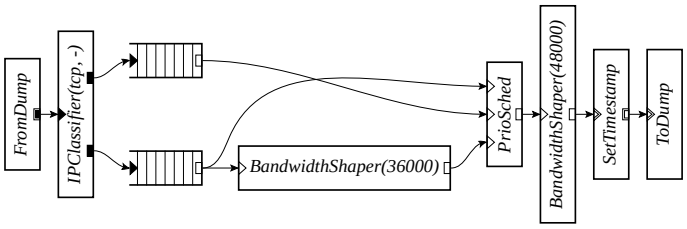

Figure 1: Problem 7 solution

```
  FromDump(f7a.dump, STOP true, TIMING true)
     -> proto :: IPClassifier(tcp, -)

  p :: PrioSched
    proto[0] -> tcp_q :: Queue(10000) -> BandwidthShaper(36000) -> [0] p
    proto[1] -> udp_q :: Queue(10000) -> [1] p
    tcp_q -> [2] p
  p -> BandwidthShaper(48000) -> SetTimestamp -> ToDump(f7b.dump, ENCAP IP)

  DriverManager(wait_stop, wait 2s)
```

See Figure 1.

***8. More error handling** [2 points]. Add error handling à la Problem 4 to the configuration of Problem 7. The router configuration file should be called `prob8.click`, it should read from '`f8a.dump`' and write to '`f8b.dump`', it should discard IP TTL-expired packets rather than generate errors, and '`f8.drops`' should report any packets dropped from queues after the errors already mentioned.*

Note the use of component element names, such as '`e/ctcp`', in the *DriverManager* configuration.

```
elementclass ErrorChecker {
   input -> cip :: CheckIPHeader
      -> i1 :: IPClassifier(tcp, udp, icmp, -)
      -> ctcp :: CheckTCPHeader
      -> ttl :: IPFilter(1 ttl 0, allow all)
      -> cl :: CheckLength(1500)
      -> output;
   i1[1] -> cudp :: CheckUDPHeader -> ttl
   i1[2] -> cicmp :: CheckICMPHeader -> ttl
   i1[3] -> ttl
   ttl[1] -> cttl :: Counter -> d :: Discard
   cl[1] -> ccl :: Counter -> d
}

FromDump(f8a.dump, STOP true, TIMING true)
   -> e :: ErrorChecker
   -> proto :: IPClassifier(tcp, -)

p :: PrioSched
  proto[0] -> tcp_q :: Queue(10000) -> BandwidthShaper(36000) -> [0] p
  proto[1] -> udp_q :: Queue(10000) -> [1] p
  tcp_q -> [2] p
p -> BandwidthShaper(48000) -> SetTimestamp -> ToDump(f8b.dump, ENCAP IP)
```

```
DriverManager(wait_stop, save e/cip.drops f8.drops,
   append e/ctcp.drops f8.drops,
   append e/cudp.drops f8.drops,
   append e/cicmp.drops f8.drops,
   append e/cttl.count f8.drops,
   append e/ccl.count f8.drops,
   append q1.drops f8.drops,
   append q2.drops f8.drops)
```

***9. Trace analysis** [**6 points**]. Write an executable script called `prob9` that takes a single command line argument, a trace file; reads that trace file; and writes the following information about the trace to the standard output:*

1. *Total number of packets.*
2. *Average packet size (rounded down to the nearest integer).*
3. *Average TCP packet size (rounded down to the nearest integer).*
4. *Average UDP packet size (rounded down to the nearest integer).*

5–261. *Number of bytes sent to each of the 256 8-bit prefixes.*
(For example, packets sent to any IP address "169.x.y.z" should be counted towards the 8-bit prefix "169".)
Don't print lines where the count is zero.

262–518. *Number of bytes sent by each of the 256 8-bit subnets.*

Use an output format like this (don't include the parenthetical comments):

`10`	(Packet count)
`765`	(Average packet size)
`1500`	(Average TCP packet size)
`30`	(Average UDP packet size)
`dst 169 7650`	(Prefix 169 received 7650 bytes)
`src 18 7500`	(Prefix 18 sent 7500 bytes)
`src 19 150`	(Prefix 19 sent 150 bytes)

The most common error here was forgetting that the input stream might contain no packets, and then dividing by zero when reporting an average packet size. I also took off a point if you ran Click 500 or more times to get your result – this is very expensive, and prevents working on standard input.

```
#! /bin/sh

click -e "
FromDump($1, STOP true)
   -> c :: Counter
   -> ip :: IPClassifier(tcp, udp, -)
   -> ct :: Counter
   -> shunt ::  input -> output
   -> sa :: AggregateIP(ip src/8)
   -> s :: AggregateCounter(BYTES true)
   -> da :: AggregateIP(ip dst/8)
   -> d :: AggregateCounter(BYTES true)
   -> Discard

sa[1] -> Print -> Discard
da[1] -> Print -> Discard
```

```
ip[1] -> cu :: Counter -> shunt
ip[2] -> shunt

DriverManager(wait_stop, save c.count -, save c.byte_count -,
      save ct.count -, save ct.byte_count -,
      save cu.count -, save cu.byte_count -,
      write d.write_ascii_file -,
      save ip.config -, // get odd line
      write s.write_ascii_file -)
" | perl -e '
$cn = <STDIN>; $cn_nonz = ($cn + 0) || 1;
$cb = <STDIN>;
$ctn = <STDIN>; $ctn_nonz = ($ctn + 0) || 1;
$ctb = <STDIN>;
$cun = <STDIN>; $cun_nonz = ($cun + 0) || 1;
$cub = <STDIN>;
print $cn, int($cb/$cn_nonz), "\n", int($ctb/$ctn_nonz), "\n", int($cub/$cun_nonz), "\n";
while (($_ = <STDIN>) && $_ !~ /^t/) {
   print "dst $_" if !/^!/;
}
while (($_ = <STDIN>)) {
   print "src $_" if !/^!/;
}
'
```

***10. Element implementation** [**EXTRA CREDIT up to 6 points**]. Create an element of your own. For example, implement a scheduler, such as a Smoothed Round Robin scheduler (Guo Chuanxiong, "SRR: An O(1) Time Complexity Packet Scheduler for Flows in Multi-Service Packet Networks", Proc. SIGCOMM 2001); or an active queue management scheme, such as FPQ (Robert Morris, "Scalable TCP Congestion Control", Proc. INFOCOM 2000); or a packet source, such as an element that reads a file, then sends 1500-byte TCP packets containing that file's contents as if TCP had sent them; or a packet sink, such as an element that sends the corresponding TCP acknowledgement for each TCP packet received. (The TCP packet source and sink examples can be made pretty interesting if you implement more-or-less-correct TCP behavior.) Be creative!*

Here's the header description of Steve VanDeBogart's element.

NAME

PaintRegex – Click element; sets packet paint annotations if the body of the packet matches a regular expression

SYNOPSIS

PaintRegex(X, Regex [, *KEYWORDS*])

DESCRIPTION

Sets each packet's paint annotation to X, an integer 0..255. Note that a packet may only be painted with one color.

Keyword arguments are:

EXTENDED

Boolean. Use POSIX extended regular expressions? If false, PaintRegex uses basic POSIX regular expressions. Default: false

IGNORE_CASE

Boolean. If true, the regular expression is case insensitvie. Default: false

DOT_MATCH_CR

Boolean. If true, . (the any-char-match) will match carriage returns. Default: true

NULL_CHAR_SUB

An integer 0..255. The supporting library treats characters with value 0 as an end of string. To get around this problem, PaintRegex changes all characters with value 0 to the value specified here. Default: 1

HANDLERS

color (read/write)

get/set the color to paint

regex (read/write)

get/set the regex to match with (not SMP safe)

NOTES

The paint annotation is stored in user annotation 0.