Report of Click Labs
Network management and QoS provisioning Course

Hong-Nam Hoang, Manh-Ha Nguyen and Xuan-Thu Thi Le

February 11, 2011

Contents

1 Introduction - ClickLabs package

1.1 File organization

elements/ This directory contains all the additional click elements using in the lab.

plot-template/ This directory contains templates used for plotting data by gnuplot. These files are used by draw-graph.sh

bin/update-elements.sh Run this file to update the new elements implemented in directory elements (above). For more information, type: ./update-elements.sh -h

bin/visual-clicky.sh Shell script to visualize click experiment using clicky. For more information, type: ./visual-clicky.sh -h

bin/init.sh Initialize Click environment for lab. Just run init.sh in the first time you get this source or click source directory changed.

bin/eclick-compile.sh Extend the Click file. A click file can include another one to reuse some compound elements (similar include in C, or import in Java). File eclick-compile.sh is used to translate (or flatten) these extended-click file to a normal click file.

bin/convert-click-dump.sh This script used to transform dump files from click (binary files) into text files. Note: this is one-way transformation, the binary files cannot be recovered from the text files.

bin/draw-graph.sh This script is used to draw graphs from data extracted in CLICK dump files. Just provide the dump files, this script will generate a graph for you. Note: No need to use convert-click-dump.sh before using draw-graph.sh.

bin/draw-graph-framerelay.sh Based on draw-graph.sh, this script helps to show the characteristics of verifying a conformance flow (which is deal with CIR, CBS, EBS).

clicky.ccss File supporting Clicky Cascading Style Sheets. It controls the appearance of a Clicky diagram with style sheets written in a CSS-like language.

1-test-config/

2-tcp-udp-generation/

3-shaper-policer/

4-scheduler/

1.2 Some introductions before surfing Click configurations

1. First of all, initialize the click environment for these stuffs. Run file init.sh:
   `chmod +x init.sh`
   `./init.sh`
   Normally, init process takes long time for the first finding Click source path. To save time, you can create file `~/.clickrc` with the content similar to this:
   `export CLICK_SRC=/home/iizke/click/click-1.8.0`

2. While finishing to code some Click elements, put it in directory `elements`, and then run file `update-elements.sh` to compile and install new elements:
   `update-elements.sh`

3. Explore the click configuration by using tool `visual-clicky.sh`. Simple way to use:
   `visual-clicky.sh $CLICK_CONFIGURATION_FILE`

4. To support easy-reading and team-working activities, we developed a tool to allow including some click files into a click file. If you write some Click files as "*library*" files, you can reuse it by using *include statements*. For example, we have `TCP_Source.click` to implement a TCP-generator, and `UDP_Source.click` to implement an UDP-generator. In `TCP_UDP.click`, we reuse the implementation of these generator by adding these lines at anywhere in `TCP_UDP.click` file (but should be on the top for easy reading):
```
------------------------ file:  TCP_UDP.click ---------------------------
//include "TCP_Source.click"
//include "UDP_Source.click"
...
-------------------------------------------------------------------------
```
The syntax of include statement is simple:
`//#include "CLICK_FILE_PATH"`
where `CLICK_FILE_PATH` can be relative or absolute path. After that, you have to use our tool (`eclick-compile.sh`) to pre-compile this file before simulating it by Click, for example:
`eclick-compile.sh -o extend-TCP_UDP.click [-f] TCP_UDP.click`
Note: if using tool `visual-clicky.sh`, you don't have to pre-compile the extended-click file. It will do all automatically.

5. To visualize your packet stream at input or output, we have developed `draw-graph.sh` to generate graph as picture (using `gnuplot` that should be installed before). The second, you have to provide the data. Normally, we usually generate data from Click with element `ToDump`. This data follows the tcpdump-like format. When you get the data, the last action you need is to run this command:
```
draw-graph.sh -f dataIn.dump -f dataOut.dump
[-o PNG_FILES]
[--plot-type COUNT (default) | RATE | DENSITY]
[--xrange 233:23221] [--yrange 282:2922]
[--xlabel XYZ] [--ylabel ABC]
[--xcol 2] [--ycol 1]
```
After program `draw-graph.sh` finishes its work, it will create a picture file (PNG file). If user does not use output option (-o), this program will export to screen (using default output file `/dev/output`). You may want to change the plotting template by modify files in `plot-template` directory.

2 Test configuration

In the first time of using Click, we try to implement `Counter_test` element, `Random_IP_generator` element using basic Click elements, such as `Print, InfiniteSource, RatedSource, Script`, also trying to modify a part of source code of `InfiniteSource` to generate packets that randomize byte value at a specific location in payload.

2.1 `Counter_test` Click configuration

Location: `1-test-config/Counter_test.click`

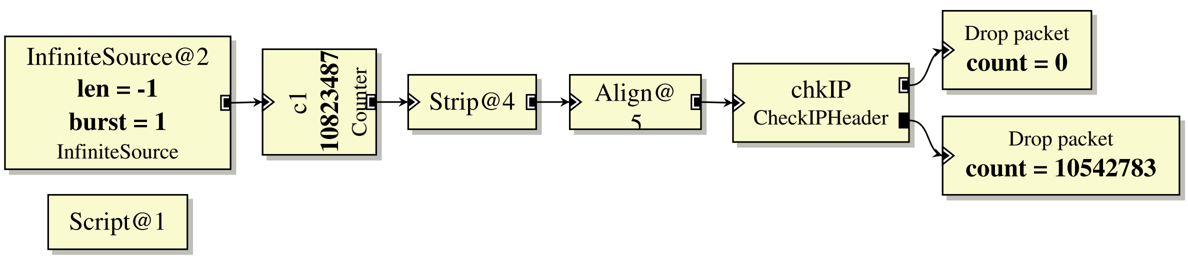


Figure 1: `Counter_test` Click configuration

To avoid IP CRC checking, we temporarily disable CRC checking by using flag "`CHECKSUM false`" in `CheckIPHeader` element. Another solution is to use `SetIPChecksum` to repair CRC in generated IP packets. Since we can visualize the result by using `clicky` to replace steps that print out screen counting results from `Counter` elements.

2.2 `RandInfiniteSource` element

Location: `elements/randinfinitesource.*`
This element is implemented by modifying source of `InfiniteSource` element. Its function is similar to `InfiniteSource`, but by adding one more keyword (`RNDBYTEID`), generated packets may have random byte value at a specified position in payload. The idea of implementation is that before pushing out packets to the output port, packet payload is changed. Originally, data packet is already prepared one time by `setup_packet` function before `InfiniteSource` releases packets. To make new element work, after modifying data, `setup_packet` function should be called, otherwise generated packets do not change their content. Figure 2 shows the result of using `RandInfiniteSource` element to generate five packets with random value at the first byte.

```
iizke@iizke-machine:~/svn/clicklabs/1-test-config$ click test-randinfinitesource.click
rand at byte 1:    8 | 68616e64 6f6d2062
rand at byte 1:    8 | 06616e64 6f6d2062
rand at byte 1:    8 | b5616e64 6f6d2062
rand at byte 1:    8 | 7c616e64 6f6d2062
rand at byte 1:    8 | 79616e64 6f6d2062
```

Figure 2: Test `RandInfiniteSource` element with 5 packets and random at the first byte

2.3 `RandomQueue` element

There are two ideas of implementing Random Queue:

- Random at input: Pushing packets at random positions in queue, but pulling out packets as FIFO queue. We try to simulate this behavior by using built-in Click elements.
- Random at output: Pushing packets in type of FIFO, but pulling out random packets in queue. We have implemented new element called `RandomQueue`.

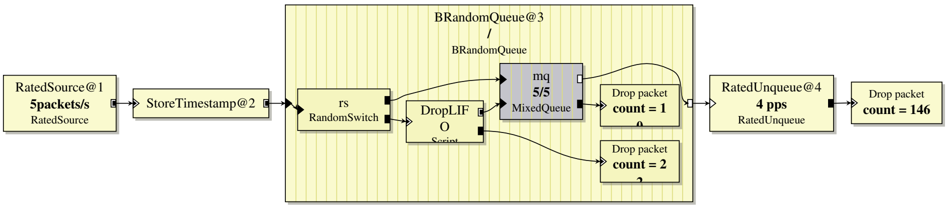
To test our element, we first generate a high rate packet at input, store current timestamps, let packet go through our elements to output which has lower rate than the input. We then print out packet timestamps to see whether they are random or not.

2.3.1 Using built-in Click elements (compound element)

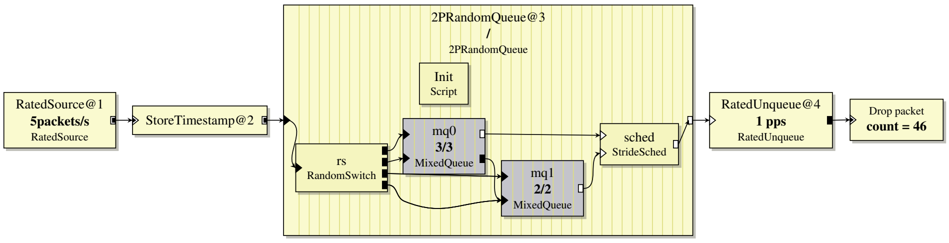
Location: `1-test-config/randomqueue.click`
We have implemented two versions:

- `BRandomQueue` (we call it Binary Random Queue): `MixedQueue` allows us to put packets in type of FIFO (input port 0) or LIFO (input port 1). Based on this function, input packets are put randomly (by `RandomSwitch` element) in either FIFO or LIFO input port. By this way, if queue size is n, there are 2^{n-1} possibilities created over total possibilities ($n!$). Technical issue: "When full, `MixedQueue` drops incoming FIFO packets, but drops the oldest packet to make room for incoming LIFO packets". It means that at that time, when observing at output, we only see packets with increasing timestamp, no randomly. We resolve this problem by writing a script to drop LIFO packet when queue is full.
- `2PRandomQueue` (Two Partition Random Queue): We expand the above idea with two queues and using Stride scheduler to join them to the output. Note that, dropped packets in the first queue are push to the second queue.

Figure 4 shows the result of testing these compound elements. The eight-byte number in each line is the timestamp of packet. We can see that this value does not increasing but randomly.



(a) `BRandomQueue`



(b) `2PRandomQueue`

Figure 3: Random Queue configurations based on built-in Click elements

```
a:    8 | 54f80c00 c69d4c4d
a:    8 | 15000000 c79d4c4d
a:    8 | d3dd0600 c79d4c4d
a:    8 | bb400900 c79d4c4d
a:    8 | 2b000000 c89d4c4d
a:    8 | e4010000 d99d4c4d
a:    8 | d0dd0600 c89d4c4d
a:    8 | 50c30000 c99d4c4d
a:    8 | 95d00300 c99d4c4d
a:    8 | 28350c00 d99d4c4d
a:    8 | 2d000000 da9d4c4d
```

(a) `BRandomQueue`

```
a:    8 | 400d0300 42884c4d
a:    8 | 36000000 41884c4d
a:    8 | 470d0300 45884c4d
a:    8 | c0270900 46884c4d
a:    8 | 00000000 48884c4d
a:    8 | 00350c00 47884c4d
a:    8 | 00000000 4a884c4d
a:    8 | c0270900 4a884c4d
a:    8 | 00000000 4c884c4d
a:    8 | 423c0c00 4c884c4d
a:    8 | 08350c00 4d884c4d
```

(b) `2PRandomQueue`

Figure 4: Test results of Random Queue configurations

2.3.2 Writing new element: `RandomQueue`

Location: `elements/randomqueue.*`
This element is inherited from `ThreadSafeQueue` class. We reuse all the source code but modifying the `pull` function to make it pull out packets randomly. Since queue data structure is not suitable for pulling out random packet (only good for the head and tail packets), we use a trick that swapping between the random packet and the first packet. Step by step in our algorithm as following:

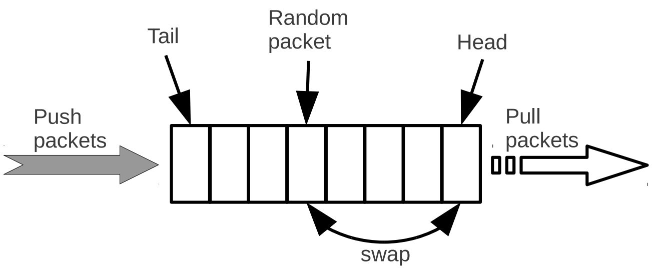


Figure 5: Behavior of `RandomQueue` element

- First, determining which packet is pulled out by a random number in the range from 0 to `RandomQueue.length`.
- Next, swapping the random packet and the first packet.
- Last, pull out the first packet (but actually the random packet).

Figure 6 shows the result of testing `RandomQueue` element. The eight-byte number in each line is the timestamp of packet. We can see that this value does not increasing but randomly. It means this element works well.

```
a:    8 | 8b350c00 a8a74c4d
a:    8 | f61a0600 a8a74c4d
a:    8 | 40d10300 a9a74c4d
a:    8 | f9dd0600 a9a74c4d
a:    8 | aee10300 aaa74c4d
a:    8 | 7d350c00 a9a74c4d
a:    8 | 86000000 aaa74c4d
a:    8 | 8ff10900 a9a74c4d
a:    8 | fb1a0600 aaa74c4d
a:    8 | 10eb0900 aaa74c4d
a:    8 | 74350c00 aaa74c4d
```

Figure 6: Test `RandomQueue` element

2.4 `Random_IP_generator` configuration

Location: `1-test-config/Random_IP_generator.click`
We combine `RandInfiniteSource, RandomQueue` with other elements to build this configuration:

- `RandInfiniteSource`: generate packets with random source IP address in the form 192.168.1.x. In this situation, we set up "RNDBYTEID 30".
- `RandomQueue`: pull out packets at random position in queue. We can replace `RandomQueue` to another types of queue, such as FIFO or LIFO, by using `MixedQueue` element (comment lines in `Random_IP_generator` configuration).
- `SetCRC32, CheckCRC32`: set or check CRC32.
- `RandomBitError`: to simulate an error free link via a queue element.
- `Script`: we add some scripts to check states:
  - `autoupdate_lostp_estimation`: calculation based on bit error from `RandomBitError` and number of 'input' packets (`c1` in figure 7).
  - `autoupdate_real_bit_error`: based on `c1, c2`.
  - `autoupdate_lostp_percent`: based on `c1, c2`.

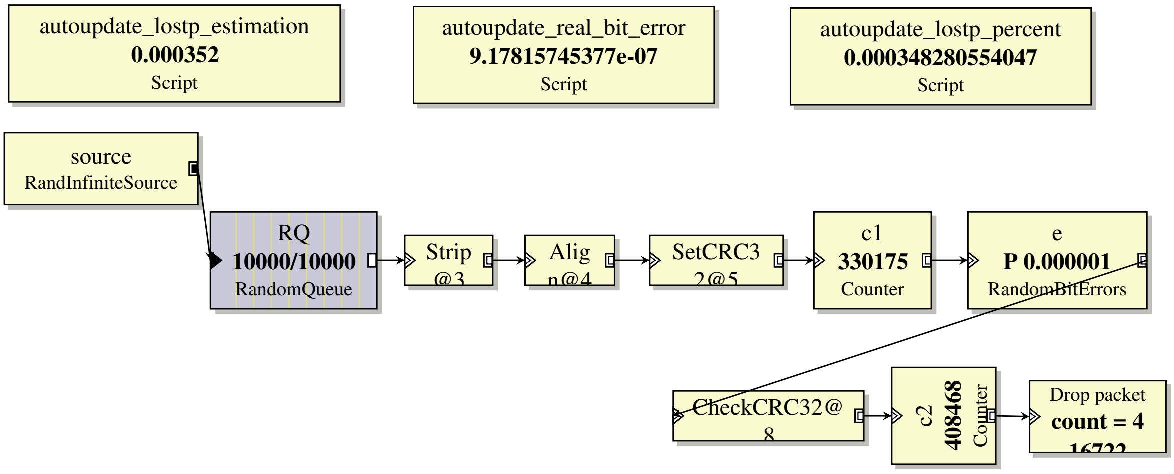


Figure 7: `Random_IP_generator` configuration

3 TCP/UDP traffic generation

3.1 TCP traffic

Location: `2-tcp-udp-generation/TCP_Source.click`
The procedure of generating TCP packet as following:

- First, we use `TimedSource` to generate TCP packet without IP header.
- After that, this packet is encapsulated IP header by `IPEncap`. Remember to setup "PROTO 0x06" to say that it is TCP packet.
- Encapsulate ethernet header with "ETHERTYPE 0x0800" in each packet.

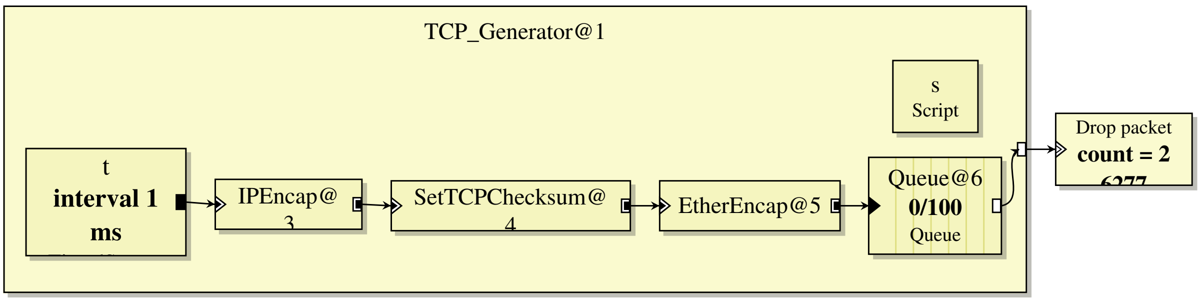


Figure 8: `TCP_Source` element

3.2 UDP traffic

Location: `2-tcp-udp-generation/UDP_Source.click`
`UDP_Generator` operates like `TCP_Generator`. Note: when using `IPEncap`, setup "PROTO 0x11" for UDP packet.

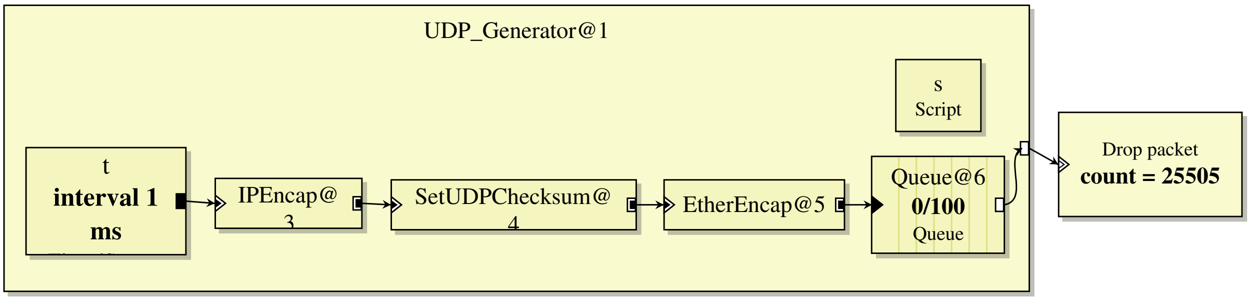


Figure 9: `UDP_Source` element

3.3 `TCP_UDP_generator` configuration

Location: `2-tcp-udp-generation/TCP_UDP_generator.click`
We build this configuration as in figure 10. TCP source is created with rate about 1000 packets per second (pps) while UDP packet rate is 1 pps. Script `autoupdate_scale` is used to check online the ratio between number of TCP packets and number of UDP packets. We see that this generator works well when both queues are not full or speed of output link (`TimedSink`) is very fast. When queues are full, the expected ratio is not guaranteed. At our observation, we try to formalize the ratio as following:

- Let r_{tcp}, r_{udp}, r are respectively the rate of TCP source, UDP source and output link. Let *ratio* is the ratio between number of TCP packets over number of UDP packets at output link.
- Let $R = (r_{tcp} + r_{udp})$, and $m = \min(r_{tcp}, r_{udp})$.
- If $r \geq R$: $ratio \to \frac{r_{tcp}}{r_{udp}}$
- If $r \leq 2m$: $ratio \to 1$
- If $2m < r < R$: $ratio \to \frac{m}{r-m}$

In general, we have a formular of ratio like: $ratio = \frac{m}{\min(R,\max(2m,r))-m}$.

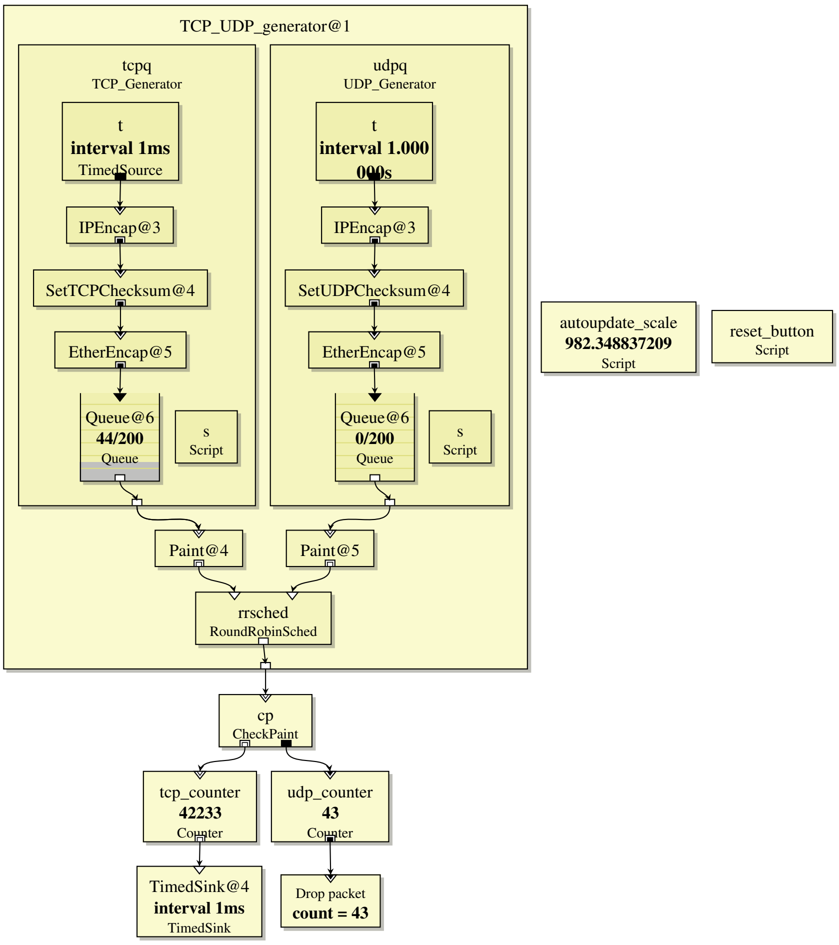


Figure 10: `TCP_UDP_generator` configuration

4 Shapers and Policers

In this part, we implement elements with consideration at packet level, not byte level.

4.1 Uncontrolled flow

Location: `3-shaper-policer/uncontrol-flow.click`
We have tried some implementations of uncontrolled flow but the main idea is that the inter-time (interval) between two consecutive packets is a random number. All implementations of uncontrolled flow are put in `3-shaper-policer/uncontrol-flow.click`. Normally, we use `RatedSource` or `TimedSource` to generate packets at a specific rate. After some time, we use Script to change their rate or interval at a random values. We list here with a few lines of description of each implementation:

- `UncontrolledFlow0`: We use two rated sources, one for generating regular rated source, one for generating burst. These sources are connected to a pull switch to choose from which source packets are generated. At any time generating packets, we choose a source to generate next packets through a script.
- `UncontrolledFlow1`: Only one source (InfiniteSource) is used and connected directly to the output. We used another two scripts named `change_rate` and `autoupdate_change_burst` to change rate and burst at runtime.
- `SimpleUncontrolledFlow`: we use `RatedSource` instead of InfiniteSource and one script to change the rate of source each one second. Script operates in type ACTIVE.

- `ProbUncontrolledFlow` (Figure 11): similar to `SimpleUncontrolledFlow` but change-rate script operates in type PACKET. When one packet go through this script, it will decide whether rate of source is changed or not based on a probability defined by user.
- `BurstUncontrolledFlow`: We use eight sources (`RatedSource`), all connected to a `ThreadSafeQueue`. At each source, packets can be dropped at a defined probability. As a result, this compound element generates packets at a particular rate but random burst (maximum burst size is 8).

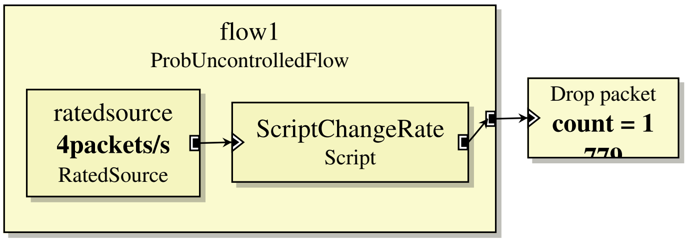


Figure 11: Uncontrolled flow with probability of changing rate source.

4.2 Leaky bucket

Location: `3-shaper-policer/leaky-bucket.click`
Since leaky bucket does not admit any burtiness, we design the leaky bucket policer with a queue of size 1 (maximum 1 packet in a queue at a time) (figure 12). After that, we use `RatedUnqueue` or `TimedSource` to create CBR source. We use both these elements because of a technical issue: when the rate is less than 1000 packets/second, `RatedUnqueue` can release packets from queue with burstiness. So, at the beginning of starting leaky policer, the `Init` script will decide which one of unqueue elements is used.

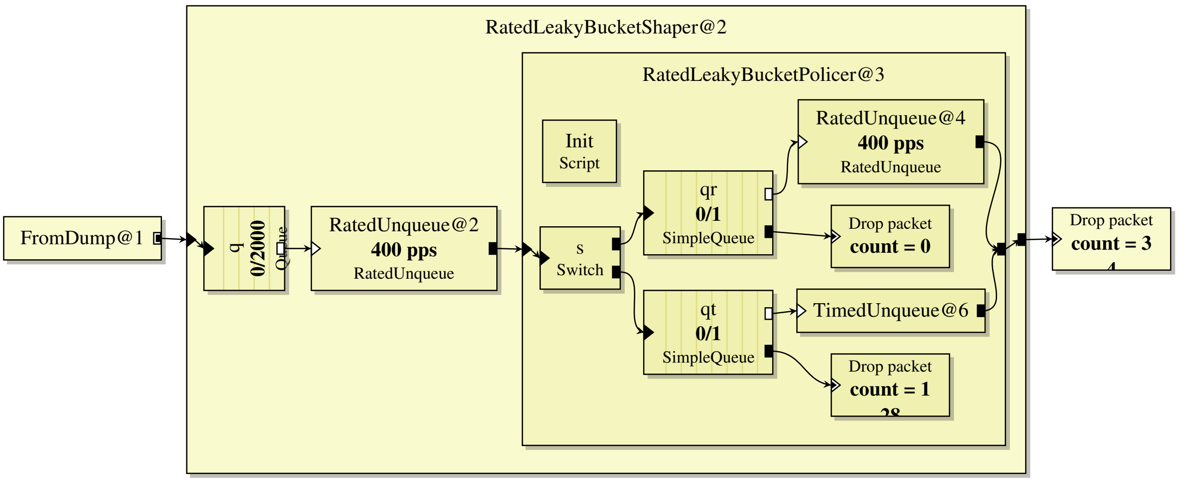


Figure 12: Leaky bucket configuration (policer and shaper)

Scenario of testing leaky bucket: `ProbUncontrolledFlow` is the source of packets with maximum rate 1000 pps (packets per second). The leaky policer only allows 400 pps. Shaper of leaky bucket uses a buffer of 2000 packets and generate packets from buffer at rate 400pps to the leaky policer. Figure 13 shows that the number of output packets is linear to time although the input is not.

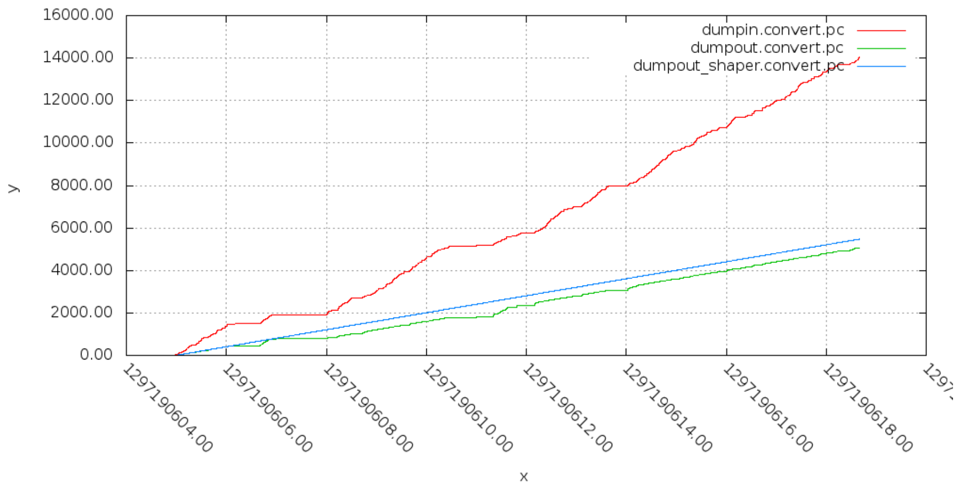


Figure 13: Monitor number of packets at input (uncontrolled flow) and output of leaky policer and shaper

4.3 Token bucket

Location: `3-shaper-policer/token-bucket.click`
Token bucket is designed similar to leaky bucket but expand the size of queue as a number of burst duration, see `RatedTokenBucketPolicer3` in figure 14. But before implementing this element, there are some alternative implementations which are more complex:

- `RatedTokenBucketPolicer1`: We use a variable-size queue in this element. The size of queue is increase with the rate that is equal to the rate of token bucket. Each time a packet goes out of queue, the capacity of queue is decrease one. To allow repeat burst at any interval time, this element uses flag REPEATED.

- `RatedTokenBucketPolicer2`: This element use a sample source (same rate as token bucket, operating as a token generator) and two counter to count the number of output packets (CI) and the number of generated token packets (CT). This element guarantees that: $CI \leq CT \leq CI + burst$. If this condition is satisfied, input packets are pushed to output immediately, otherwise they are dropped. This element releases packets to output as soon as possible (there is no queue to store input packets) while other implementations try to store input packets first and then regulate the output flow at a specific rate.

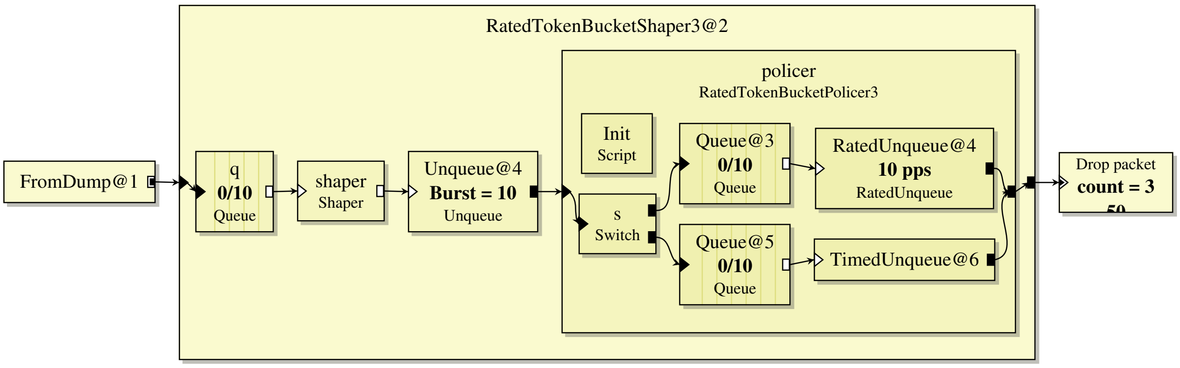


Figure 14: Token bucket configuration `RatedTokenBucketShaper3`

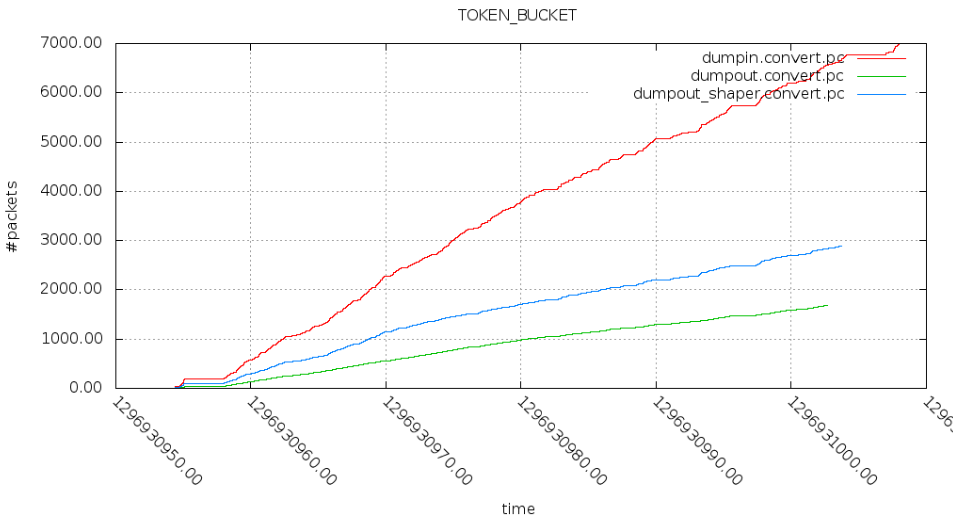


Figure 15: Monitor number of packets at input (uncontrolled flow) and output of token policer and shaper

Figure 16 does a comparison of token and leaky bucket. The scenario is: we try to regulate a `BurstUncontrolledFlow` (rate is 1 pps and maximum burst is 8) by a token and a leaky bucket policer. Token bucket policer is `RatedTokenBucketPolicer3` (rate is 10 pps, burst is 10), and leaky policer is `RatedLeakyBucketPolicer` (rate is 10 pps). We see that token policer can allow some burst duration but leaky policer limits it.

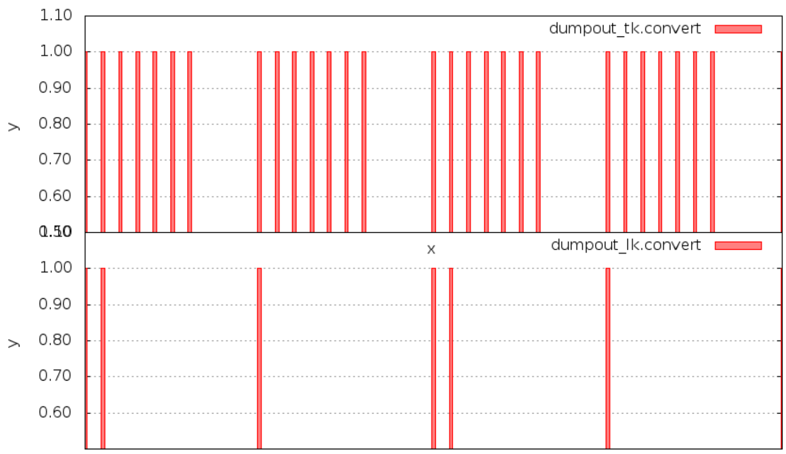


Figure 16: Comparision of Token bucket and Leaky bucket

4.4 Negotiation (CIR, CBS, EBS)

Location: `3-shaper-policer/negotiation.click`
Figure 17 shows one implementation of negotiation (`RatedNegotiablePolicer2`). The idea is: input packets are gone through `RatedLeakyBucketShaper` (with leaky rate $R_L = \frac{CIR*(CBS+EBS)}{CBS}$) and then gone through `RatedTokenBucketShaper2` (with token rate $R_T = CIR$ and burst duration is EBS). At output, we guarantee that in interval time $T = \frac{CBS}{CIR}$, maximum number of output packets is $(CBS + EBS)$ and flow is shapped to rate CIR with a maximum burst duration EBS.

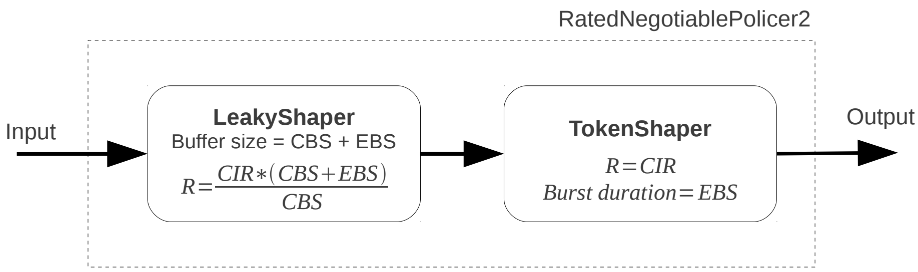


Figure 17: Implementation of `RatedNegotiablePolicer2`

We brieftly describe other implementations of negotiation which can be found in `negotiation.click` as following:

- `RatedNegotiablePolicer1`: Input packets are stored in a queue with length $(CBS + EBS)$. At each interval $T = \frac{CBS}{CIR}$, all packets in queue are released. This implementation is simple with only one `TimedUnqueue` to control T, one Queue to do negotiation. To recognize high and low priority, at each packet, before storing it in queue, we check if queue length is larger than CBS then its priority is low. However, this element is a non-work-conserving element, so we only consider it when T is small (should be $T \leq 1$, or the best case is $CBS = 1$).

- `RatedNegotiablePolicer4`: See figure 18. Input packets are classified into low and high priority. Packet is high priority and put into `HighQueue` if there is free space in `HighQueue` (capacity CBS), otherwise it is in `LowQueue` (capacity EBS) with low priority. Since rate CIR is fixed, the window time $T = \frac{CBS}{CIR}$ is scale to CIR. We use a counter `TimeCount` to know when the window time is end (by observing `TimeCount` $= CBS$). If this happens, we reset all the counters for a new window time. Since we calculate the window time base on CBS, we have to make sure that there are always packets to `TimeCount` at rate CIR. So, we use `SampleSource` generating packets at rate CIR and `PrioScheduler` to guarantee that condition. At output, there are only $(CBS + EBS)$

packets in a window time T. The high priority packets are on port 0, and the low priority packets are on port 1.

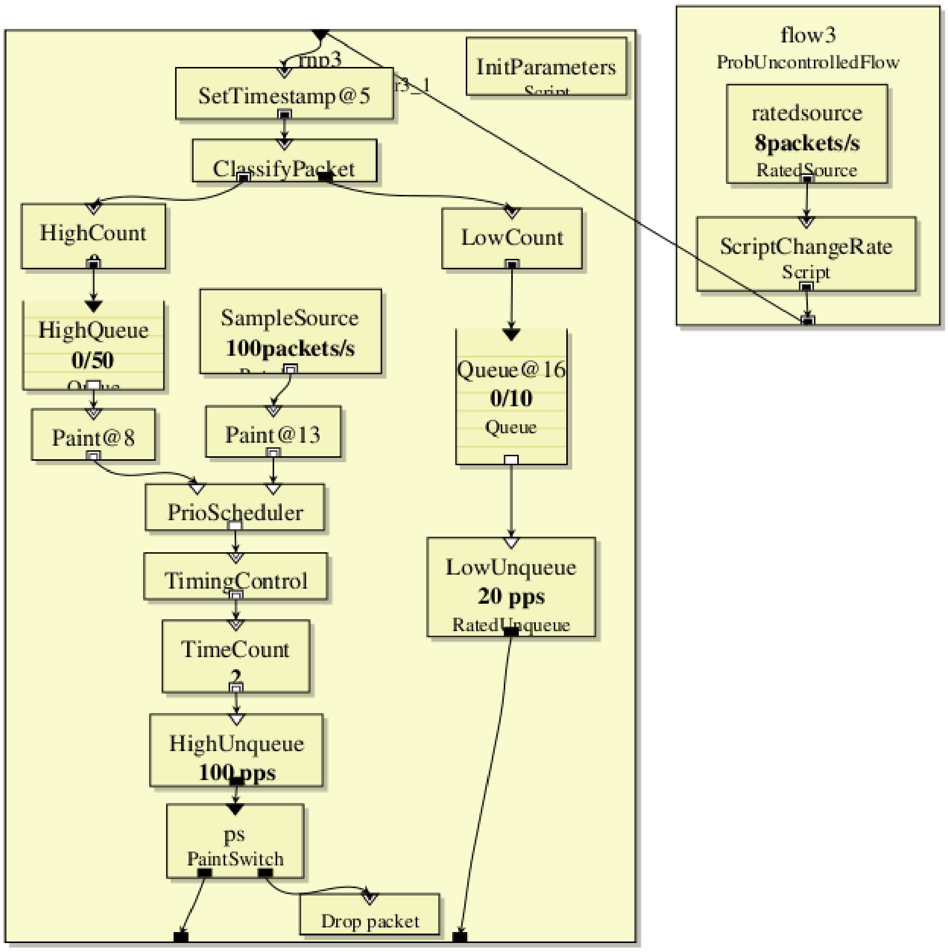
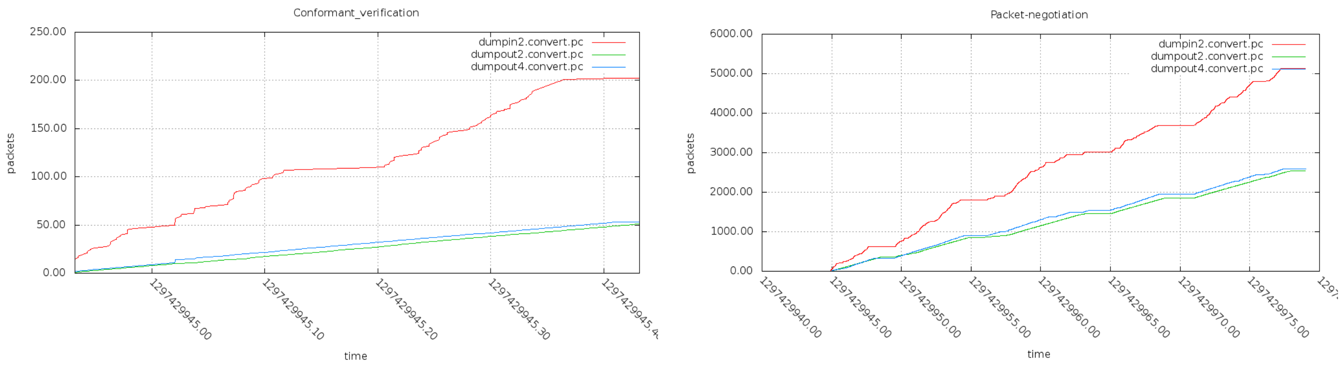


Figure 18: Implementation of `RatedNegotiablePolicer4`

We test `RatedNegotiablePolicer2` and `RatedNegotiablePolicer4` with an uncontrolled input source, maximum rate 500 pps. The negotiation is: CBS = 50, EBS = 10 and CIR = 100 (pps). Results are shown in figure 19a. The red line (`dumpin2`) is presentation of input flow. The green line (`dumpout2`) is presented `RatedNegotiablePolicer2` and the other line of `RatedNegotiablePolicer4`. to We can see that the number of output packets, in a window time $T = 0.5s$, is trimmed at 60 packets follow rules of negotiation.

4.5 Generic Cell Rate Algorithm - GCRA

Location: `3-shaper-policer/gcra.click`
GCRA configuration is in figure 20. We implement this element by using `SetVirtualClock`. Element `SetVirtualClock` works as following: when a packet comes to this element, it will set this packet's timestamp annotation to a new value, and also remember the last theoritical cell arrival time (TAT). Before letting packets approach `SetVirtualClock`, packets have to go through script `CheckTime`. If packet comes too soon, it will be dropped, otherwise it will go to `SetVirtualClock` and make a new TAT. We test this element by a `ProbUncontrolledFlow` with maximum rate 10 pps. Our `GCRA` is set up to allow $TAT = 0.2$ and $\tau = 0$. The result is shown in figure 21. This figure represents the packet arrival time of input (lower part) and output. As we can see, output flow is less dense than input flow, and the inter-time between packets is guaranteed.



(a) Number of packets in a time interval CBS/CIR

(b) Number of packets over time

Figure 19: Negotiation with `RatedNegotiablePolicer2` and `RatedNegotiablePolicer4`

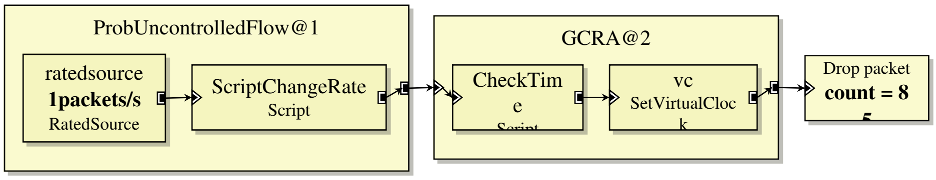


Figure 20: `GCRA` configuration

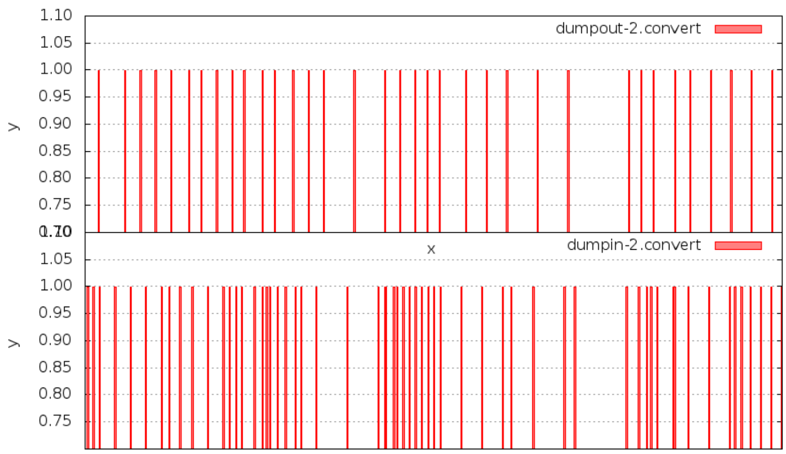


Figure 21: Testing GCRA configuration with flow `ProbUncontrolledFlow`

5 Schedulers

5.1 FIFO scheduler

Location: `4-scheduler/FIFO_Sched.click`
There are two ways of building FIFO scheduler. The first and simple way is to use `ThreadSafeQueue` element. All inputs are connected into input of queue and output of queue is connected to output of

FIFO scheduler. The second way is to use `TimeSortedSched` element to which all inputs connect through queues. We test our FIFO scheduler with three flows and their rate respectively 1 pps, 3 pps, 6 pps. FIFO scheduler only processes 1 pps. Figure 22 shows the result that the high rate flow (blue color - out2) makes a greate effect on the output link.

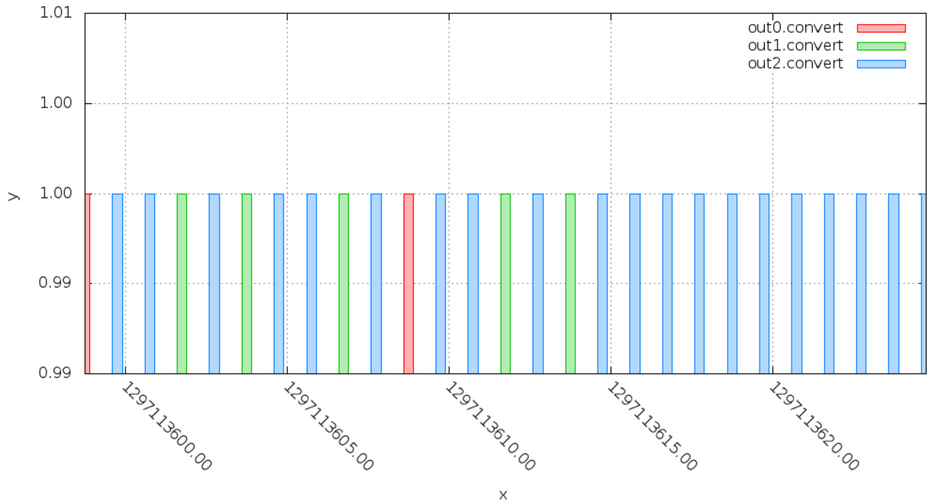


Figure 22: Testing `FIFOSched` configuration

5.2 Round Robin scheduler

Location: `4-scheduler/RR_Sched.click`
Round Robin Scheduler is built-in element in Click. We test this scheduler with the same scenario as described in testing FIFO scheduler. Although the inputs have diffirent rates, the output link share fairly to all three flows (figure 23).

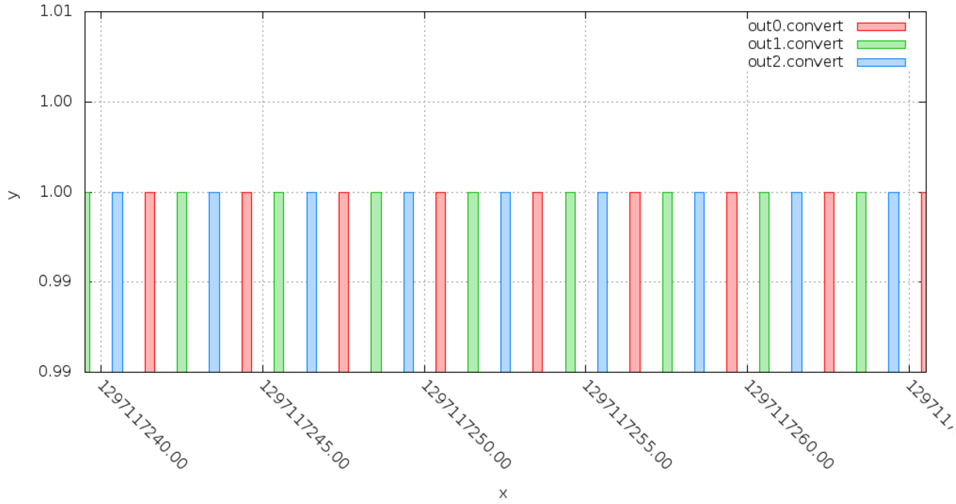


Figure 23: Testing `RRSched` configuration

5.3 Deficit Round Robin scheduler

Location: `4-scheduler/DRR_Sched.click`
Deficit Round Robin Scheduler is built-in element in Click. We also test DRR scheduler with three flows, but the size of packets in each flow are 500, 1000 and 1500 bytes respectively. The result in figure 24 shows that DRR scheduler guarantee bandwidth provided to each flow. The small-packet size flow is scheduled more times than other flows.

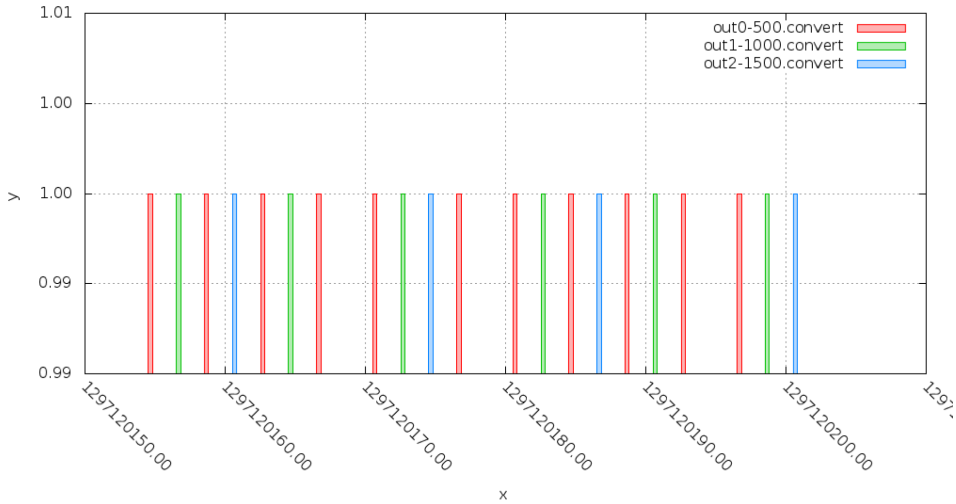


Figure 24: Testing `DRRSched` configuration

5.4 Weighted Round Robin Scheduler

We implement this scheduler in two versions. The first is compound elements that based on Round Robin scheduler. The second is a new element.

5.4.1 WRR scheduler - compound element

Location: `4-scheduler/WRR_Sched.click`
This scheduler is based on Round Robin scheduler. The main idea is that: weight of each flow is scaled to number of ports assigned to that flow. A high weight flow will have more input ports of Round Robin scheduler. In practice, we need to take care of distribution of ports to have the fairness in response time. We test WRR scheduler by using three input flows with weight 1, 2 and 3 respectively. The result is shown in figure 25.

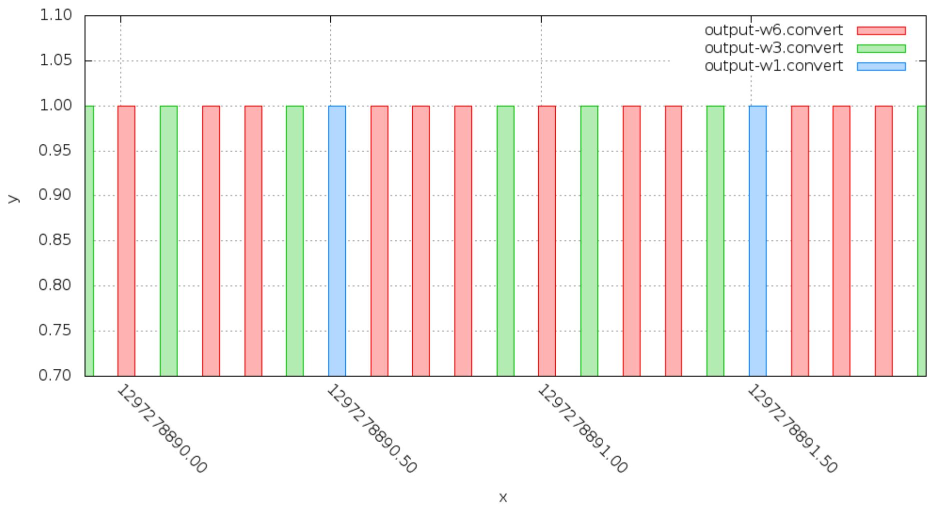


Figure 25: Testing `WRRSched` compound element

5.4.2 WRR scheduler - new element

Location: `4-scheduler/WRR_Sched_element.click`
Since WRR compound element is not easy to deal with big weight although only a few number of inputs, we developed a new element for WRR scheduler. N is number of input, w_i is weight of i-th input, W is total of weights. At initial time, this element will create a list of visited ports in period of W steps, and try to make fairness in response time. After finishing to process a packet at one port, it will process packet of next port in the list created in initial time. We test with three flows, weights are 1, 2 and 3. The result is shown in figure 26.

Figure 26: Testing `WRRSched` element

5.5 Weighted Deficit Round Robin scheduler

Not implement as a new Click element but only a compound element, similar to the compound element of WRR scheduler.

5.6 `SetVirtualClock` element

It is a new Click element which is used to support Virtual Clock scheduler and Weighted Fair Queue scheduler. Its important feature is the ability of remembering the last computed value. Each time a packet arrives, it will set a new virtual time (tag) into timestamps annotation of packet. Computation of tag is:

$$F_i^k = \max(F_i^{k-1}, V(a_i^k)) + \frac{L_i^k}{r_i}$$
$$V(0) = 0$$
$$\frac{\partial V}{\partial \tau} = \frac{1}{\sum r_j}$$

5.7 Virtual Clock scheduler

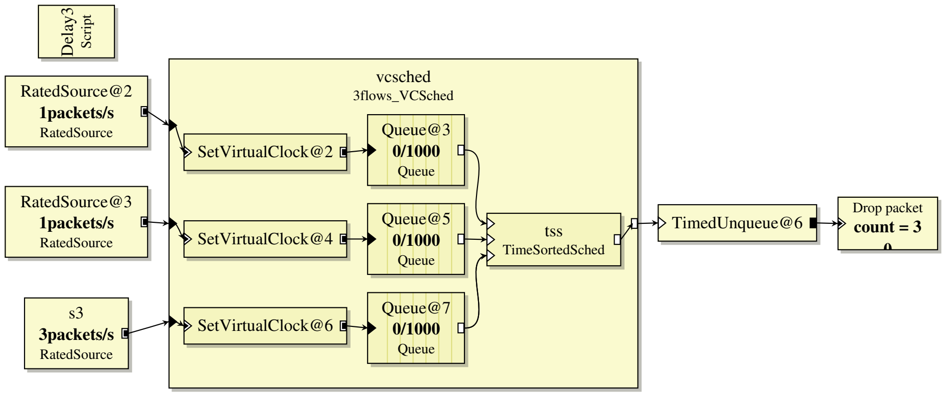


Figure 27: `VirtualClock` scheduler with 3 flows

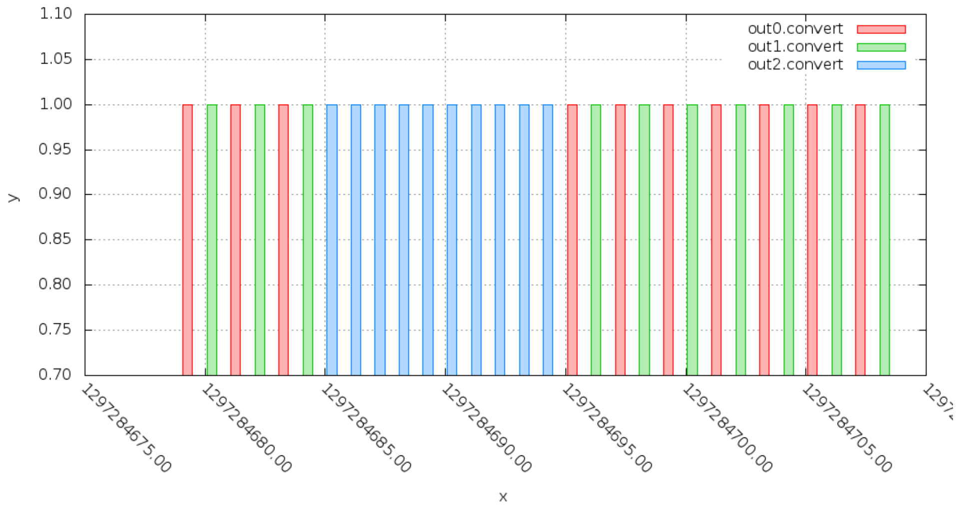


Figure 28: Testing `VirtualClock` scheduler with 3 flows

5.8 Weighted Fair Queue scheduler

6 Congestion control

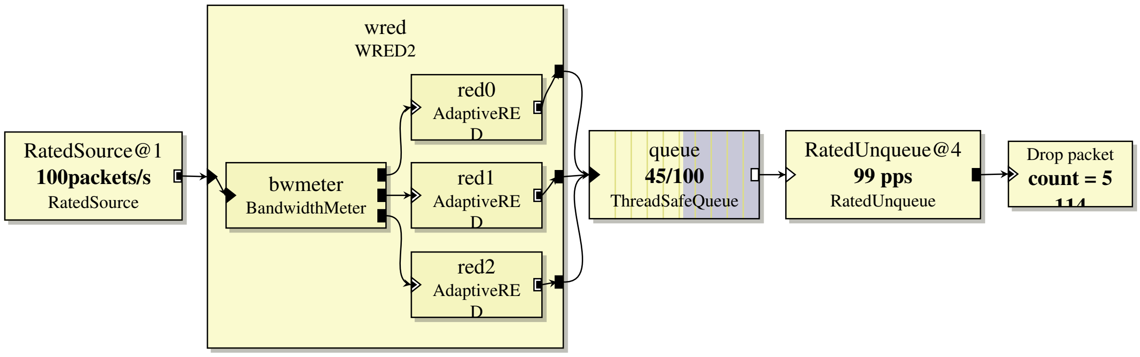


Figure 29: One simple implementation of `WRED` element