Practical report of ClickLabs
Network management and QoS provisioning Course

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

February 25, 2011

Contents

1 Introduction - ClickLabs package

ClickLabs source: `http://clicklabs.googlecode.com`.

1.1 File organization

elements/ This directory contains all the additional Click elements using in the lab. At this time, it includes:

- `randinfinitesource.{hh,cc}`: Similar `InfiniteSource` but random byte value in payload.
- `randomqueue.{hh,cc}`: Random Queue.
- `setvirtualclock.{hh,cc}`: Set virtual clock to timestamp of packet.
- `wrrsched.{hh,cc}`: Weighted Round Robin Scheduling.
- `wdrr.{hh,cc}`: Weighted Deficit Round Robin Scheduling.
- `settimestamp2.{hh,cc}`: Similar to element `SetTimestamp` with an additional keyword "`ADD`".
- `checktimestamp.{hh,cc}`: Drop packets if its timestamp is expired.
- `eddsched.{hh,cc}`: Earliest Due Date (EDD) scheduling.
- `eddqueue.{hh,cc}`: EDD Queue, used in EDD scheduling.
- `shiftqueue.{hh,cc}`: Shift Queue, used in EDD scheduling.

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 laboratory. Just run `init.sh` in the first time you get this source or whenever Click source location is 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 is 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-negotiation.sh Based on `draw-graph.sh`, this script helps to show the characteristics of verifying a conformant flow (which is a 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/ This folder contains Click configurations for the first part of Click Lab: `Counter_test`, `RandomQueue`, `RandInfiniteSource`, `Random_IP_generator`.

2-tcp-udp-generation/ This folder contains Click configurations for the part of TCP/UDP generator.

3-shaper-policer/ This folder contains Click configurations for the part of Shaper and Policer: uncontrolled flows, Leaky bucket, Token bucket, negotiation with {CBS, EBS, CIR}, GCRA.

4-scheduler/ This folder contains Click configurations for the part of Scheduler: FIFO, RR, DRR, WRR, WDRR, Virtual clock, Weighted Fair Queueing, Earliest Due Date.

5-congestion/ Click configurations for controlling congestion. At this time, there is only one Click configuration of WRED (Weighted-RED for two ranges of input bandwidth).

1.2 Some introductions or tips 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, this 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. When finishing to write some codes of Click elements (not configurations), put it in directory `elements`, and then run file `update-elements.sh` to compile and install new elements:
   `update-elements.sh`

3. Explore and test Click configurations by using tool `visual-clicky.sh`. Simple way to use:
   `visual-clicky.sh $CLICK_CONFIGURATION_FILE`
   This tool helps us to run this configurations in Click and visualize its activities by clicky. In the case of testing without clicky, for example dumping input and output packets to files, we can add option `--noclicky` to avoid using clicky (for good performance) but just transforming extended-Click file to normal Click file (if necessary) and running it in Click.

4. To support easy-reading and team-working activities, we developed a tool to allow including some Click files into one 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 do not 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. We often 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 like in figure 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.

```
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]
```

Figure 1: An example of using command `draw-graph.sh`

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`

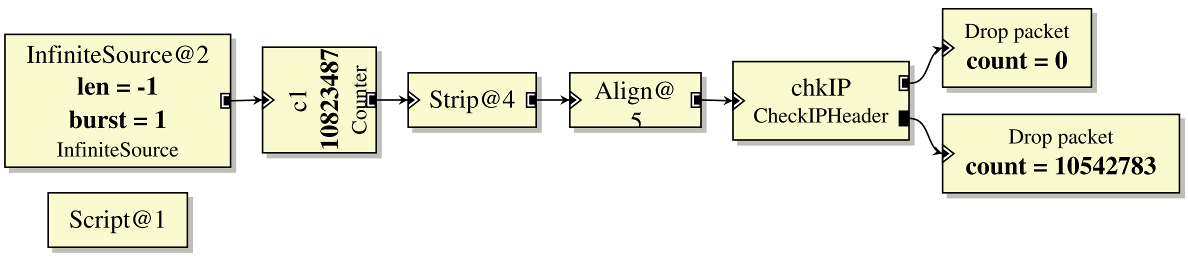


Figure 2: `Counter_test` Click configuration

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. The result is visualized by `clicky`.

2.2 `RandInfiniteSource` element

Location: `elements/randinfinitesource.*`
This element is implemented by modifying source of `InfiniteSource` element. Its function is similar to `InfiniteSource`, one more keyword (`RNDBYTEID`) is added to generate packets having 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 3 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 3: Test `RandInfiniteSource` element with 5 packets and random at the first byte

2.3 `RandomQueue` element

There are two ideas of implementing Random Queue:

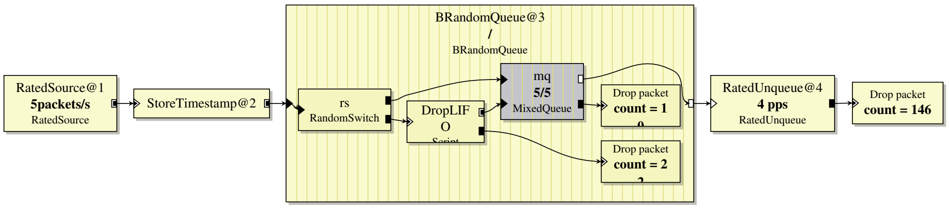
- Random at input: Pushing packets at random positions in queue, but pulling out packets as FIFO queue. We try to simulate this behavior by using built-in Click elements.
- Random at output: Pushing packets in type of FIFO, but pulling out random packets in queue. We have implemented new element called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

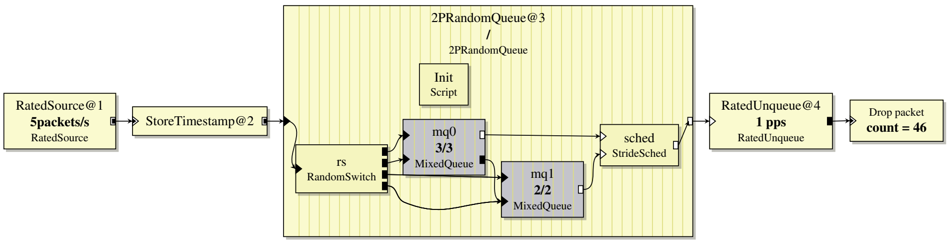
2.3.1 Using built-in Click elements (compound element)

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 ($n!$).
- `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.



(a) `BRandomQueue`



(b) `2PRandomQueue`

Figure 4: Random Queue configurations based on built-in Click elements

Figure 5 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.

2.3.2 Writing new element: `RandomQueue`

Location: `elements/randomqueue.*`
This element is inherited from `ThreadSafeQueue` class. We reuse all the source code but modify 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:

```
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 5: Test results of Random Queue configurations

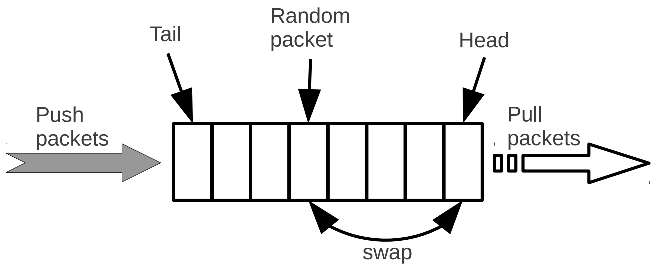


Figure 6: Behavior of `RandomQueue` element

- 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 7 shows the result of testing `RandomQueue` element. The eight-byte number in each line is the timestamp of packet. We can see that 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 7: 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 of expected lost packets over input packets, based on bit error from `RandomBitError` and number of 'input' packets (`c1` in figure 8).
  - `autoupdate_real_bit_error`: real bit error based on `c1, c2`.
  - `autoupdate_lostp_percent`: real lost packets over input packets based on `c1, c2`.

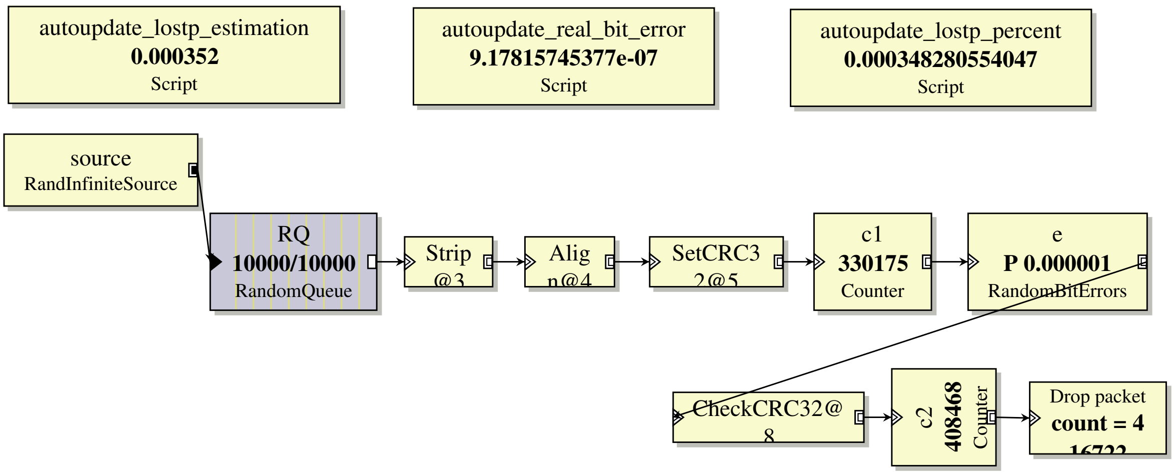


Figure 8: `Random_IP_generator` configuration

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.

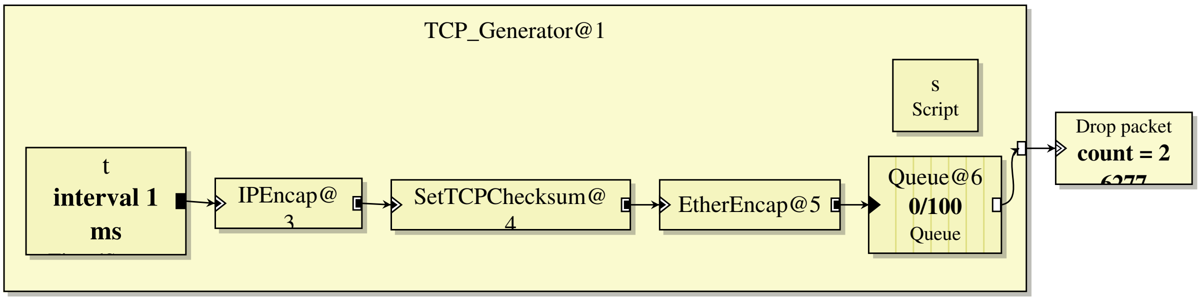


Figure 9: `TCP_Source` element

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.

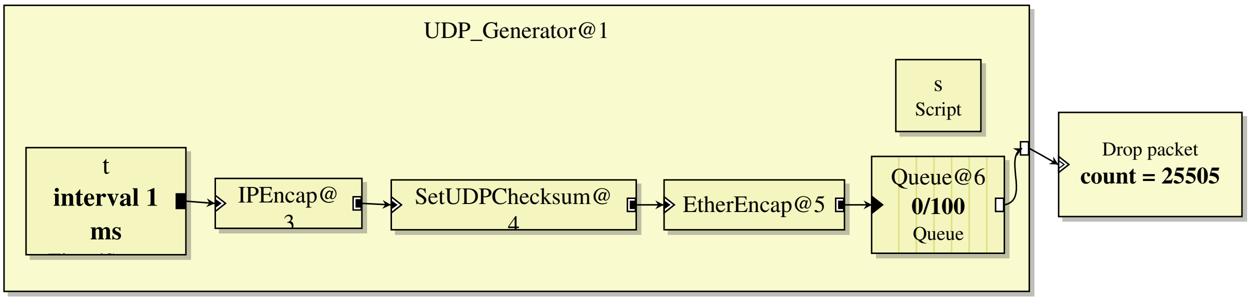


Figure 10: `UDP_Source` element

3.3 `TCP_UDP_generator` configuration

Location: `2-tcp-udp-generation/TCP_UDP_generator.click`
We build this configuration as in figure 11. TCP source is created with rate about 1000 packets per second (pps) while UDP packet rate is 1 pps. Both sources are connected to a Round Robin scheduler (`rrsched`). 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. Assume that $r_{tcp} > r_{udp}$.
- Let $R = (r_{tcp} + r_{udp})$.
- If $r \geq R$: $ratio \to \frac{r_{tcp}}{r_{udp}}$
- If $r \leq 2 * r_{udp}$: $ratio \to 1$
- If $(2 * r_{udp}) < r < R$: $ratio \to \frac{r - r_{udp}}{r_{udp}}$

Generally, we have: $ratio = \frac{\min(R, \max(2*r_{udp}, r)) - r_{udp}}{r_{udp}}$, and $r_{tcp} > r_{udp}$.

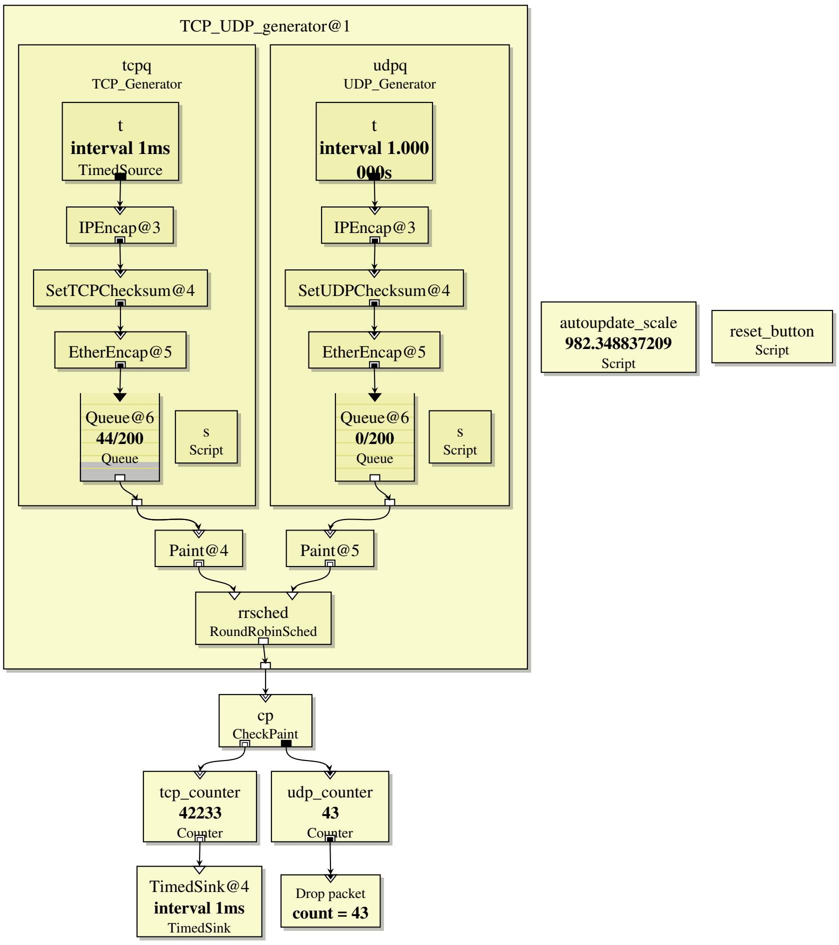


Figure 11: `TCP_UDP_generator` configuration

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 element `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 duration at runtime.

- `SimpleUncontrolledFlow`: we use `RatedSource` instead of `InfiniteSource` and one script to change the rate of source. This script operates in type ACTIVE and period of one second.

- `ProbUncontrolledFlow` (Figure 12): 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`) generating packets in the same rate, all connected to a `ThreadSafeQueue`. In 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 duration (maximum is 8).

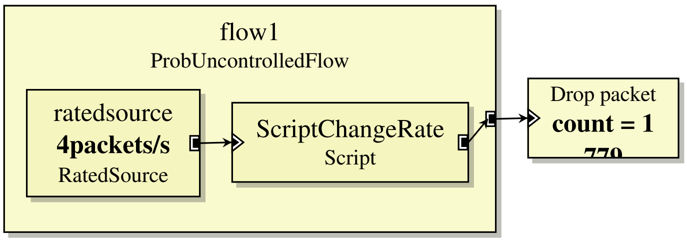


Figure 12: Uncontrolled flow with probability of changing rate source.

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 one (maximum one packet in a queue at a time) (figure 13). 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.

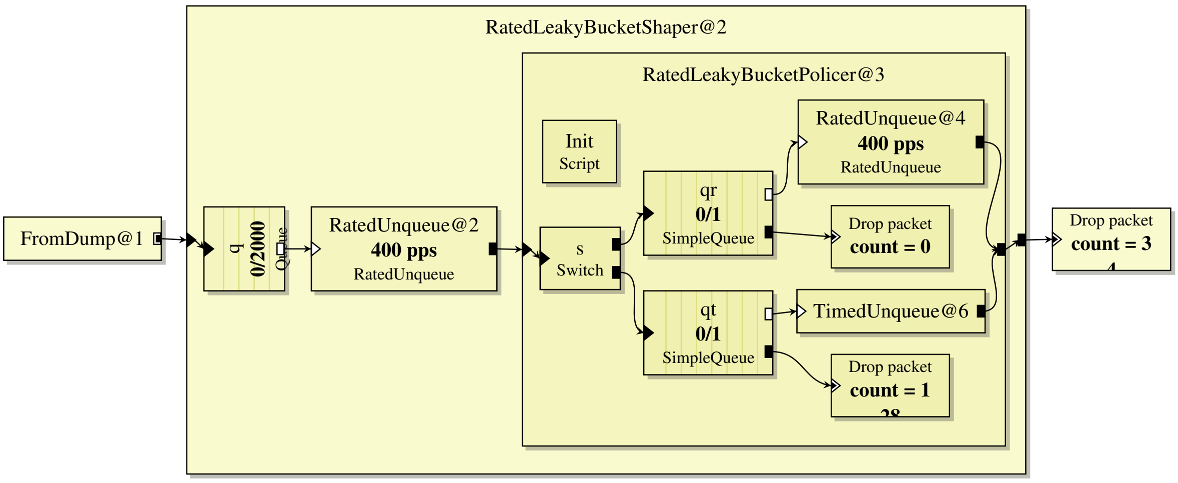


Figure 13: Leaky bucket configuration (policer and shaper)

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 14 shows that the number of output packets is linear to time although the input is not.

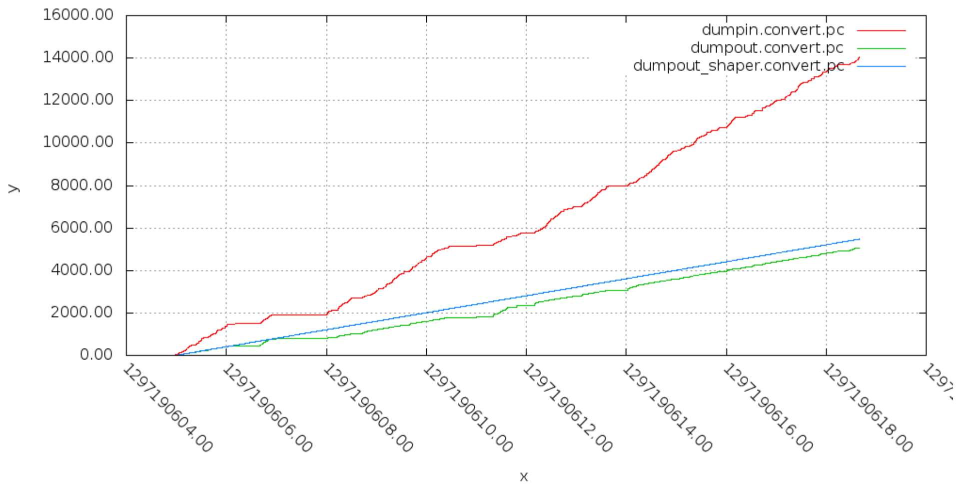


Figure 14: Monitor number of packets at input (uncontrolled flow) and output of leaky policer and shaper

4.3 Token bucket

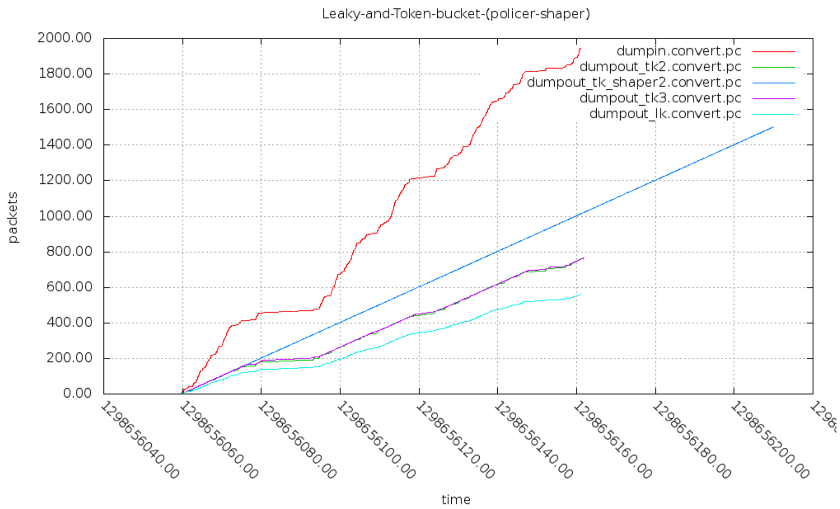
Location: `3-shaper-policer/token-bucket.click`
Token bucket is designed similar to leaky bucket but expand the size of queue as a number of burst duration, see `RatedTokenBucketPolicer3` in figure 15. The issue in this configuration is that it does not allow burst at output but allow some burst at input. 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 increased with the rate that is equal to the rate of token bucket. Each time a packet goes out of queue, size of queue is decreased one. To allow repeating burst at any periodic interval time, this element uses flag REPEATED.

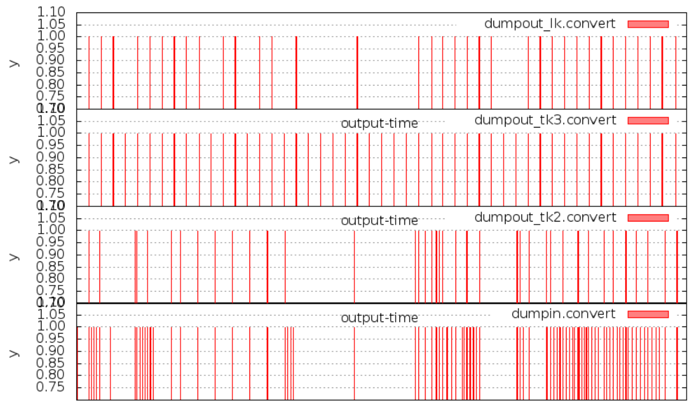
- `RatedTokenBucketPolicer2`: This element uses a sample source (same rate as token bucket, operating as a *token generator*) and two counters counting the number of output packets (CO) and the number of generated token packets (CT). This element guarantees that at any time,

$$CO \leq CT \leq CO + burst\_duration$$

  If it is satisfied, input packets are pushed to output immediately, otherwise they are dropped. This element releases packets to output as soon as possible (no queue is needed to store input packets) while other implementations try to store input packets first and then regulate the output flow at a given rate.



(a) Number of packets over time



(b) Distribution of packet at output

Figure 16: Test results of using leaky and token bucket to police and shape flows

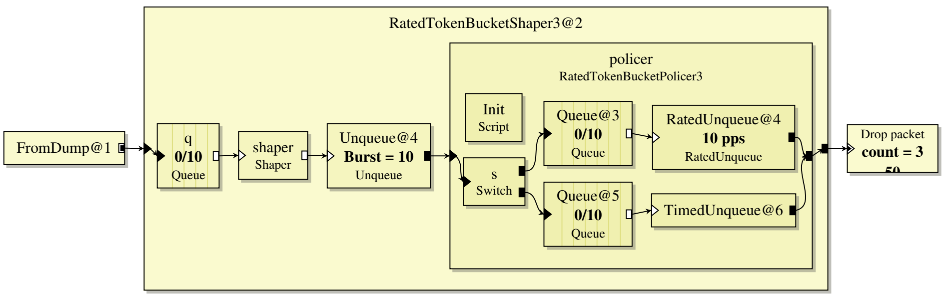


Figure 15: Token bucket configuration `RatedTokenBucketShaper3`

Figure 16 does a comparison of token and leaky bucket. The scenario is: we try to regulate a `ProbUncontrolledFlow` (maximum rate is 60 pps, probability of changing rate is 0.7) by token and leaky bucket. We test two configurations of token bucket: `RatedTokenBucketPolicer2` and `RatedTokenBucketPolicer3` (rate is 10 pps, burst is 10). Leaky bucket is `RatedLeakyBucketPolicer` (rate is 10 pps). All the shapers have buffer of 500 packets. From the result, figure 16(a) shows the number of packets going out at output for each policers and shapers. Since all shapers work similarly in this case, we only plot `RatedTokenBucketShaper2`. We can see that two configurations of token bucket work similarly. But in figure 16(b), wee see the differences between them when looking at the distribution of output packets over time.

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 shaped to rate CIR with a maximum burst duration EBS.

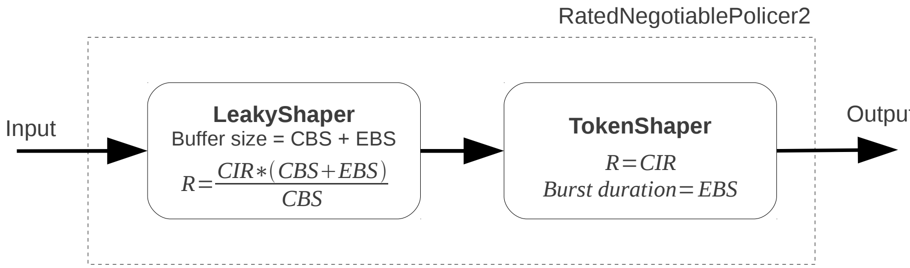


Figure 17: Implementation of `RatedNegotiablePolicer2`

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 CBS. 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 based on CBS, we have to make sure that there are always packets to `TimeCount` at rate CIR. So, `SampleSource` is used to generate packets at rate CIR to guarantee that condition. Whenever there are no packets from `HighQueue`, the `PrioScheduler` will get packets from `SampleSource`. In the end, temporary packets will be removed before going to the output. The number of packets in a window time T always less than or equal to $(CBS + EBS)$. The high priority packets are on port 0, and the low priority packets are on port 1.

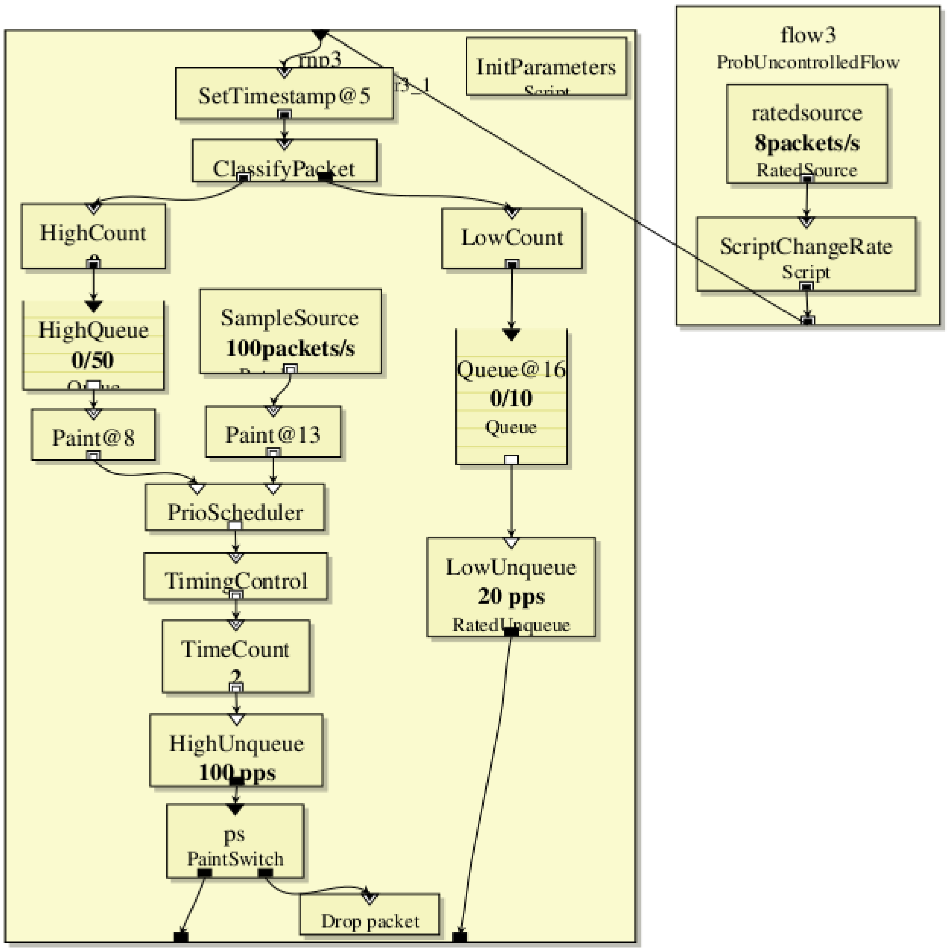
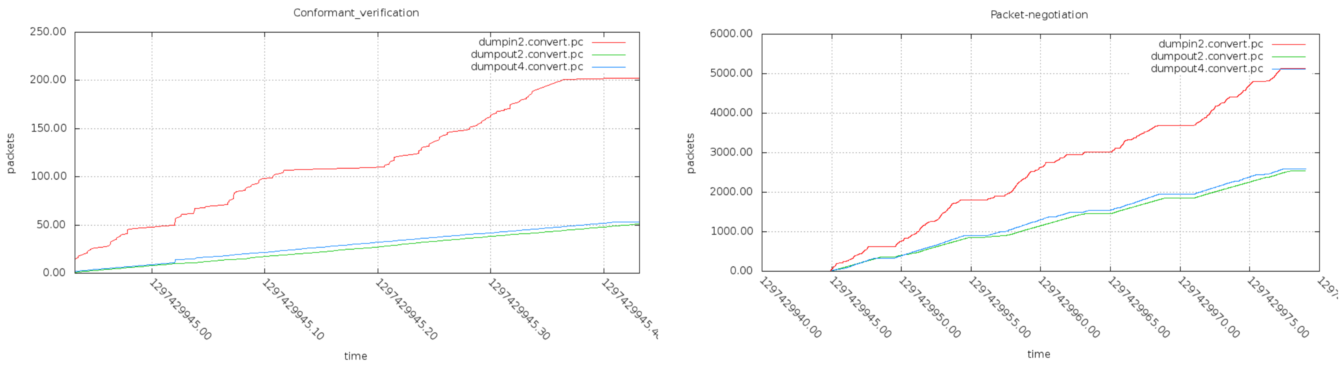


Figure 18: Implementation of `RatedNegotiablePolicer4`

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 19. The red line (`dumpin2`) is representation of input flow. The green line (`dumpout2`) is represented to `RatedNegotiablePolicer2` and the other line of `RatedNegotiablePolicer4` (`dumpout4`). We can see that the number of output packets, in a window time $T = 0.5s$, is trimmed at 60 packets following the rules of negotiation, but line `dumpout4` is a little higher than line `dumpout2`.

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 (more details in section 5.6): 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 $\tau = 0$. In figure 21, we plot the packet arrival time of input (lower part) and output and see that output flow is less dense than input flow, and the inter-time between packets is guaranteed.



(a) Number of packets in a time interval $T = \frac{CBS}{CIR} = 0.5s$

(b) Number of packets over time

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

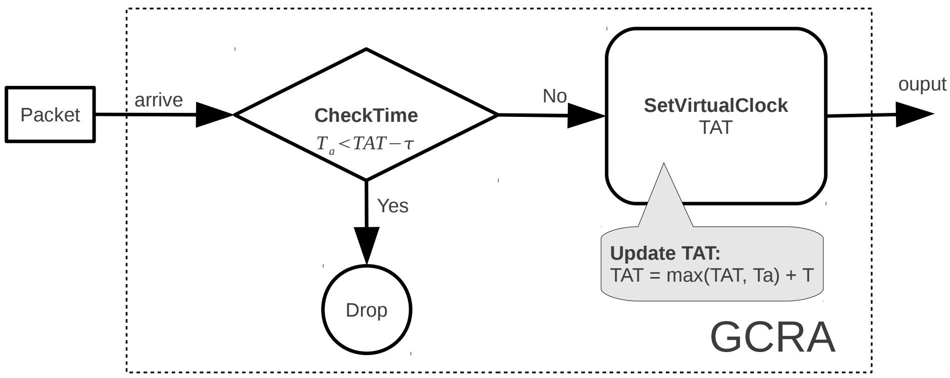


Figure 20: `GCRA` configuration

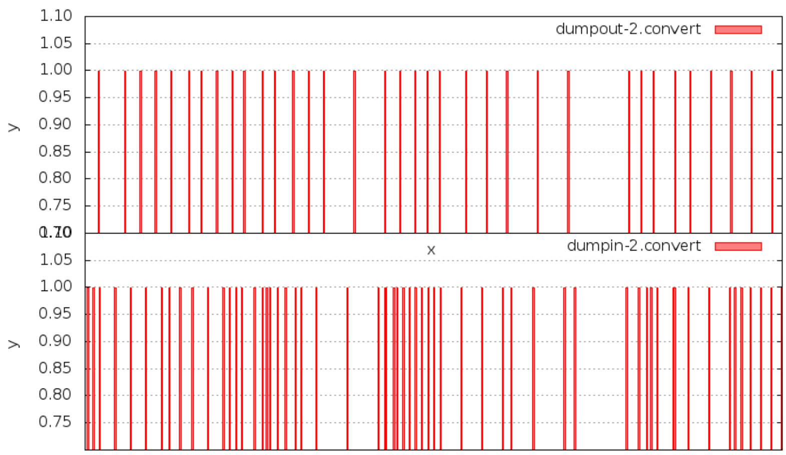


Figure 21: Testing GCRA configuration with flow `ProbUncontrolledFlow`

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 huge effect on the output link.

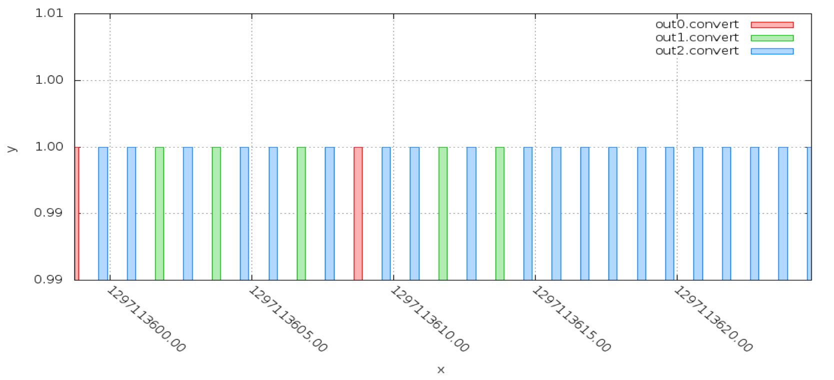


Figure 22: Testing `FIFOSched` configuration: distribution of packet at output over time

5.2 Round Robin scheduler

Location: `4-scheduler/RR_Sched.click`
Round Robin Scheduler is a 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 is shared fairly to all three flows (figure 23) but it does not take care of packet's length.

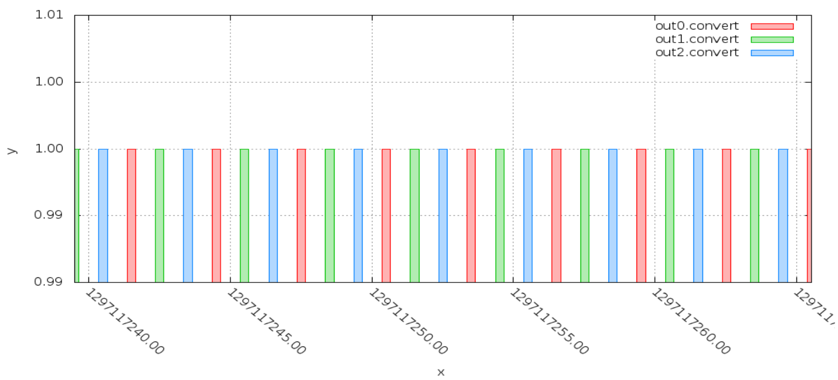


Figure 23: Testing `RRSched` configuration: distribution of packet at output over time

5.3 Deficit Round Robin scheduler

Location: `4-scheduler/DRR_Sched.click`
Deficit Round Robin (DRR) Scheduler is a 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. When working with DRR scheduler and checking its source code, we know that the quantum byte is 500. It means for each round, deficit of each flow is increased 500. If we setup flows with too small in packet length, those flows will send out a lot of packets before next flows are scheduled to send packets. Or in contrast, flows having large size packets will wait for a long time before sending out packets.

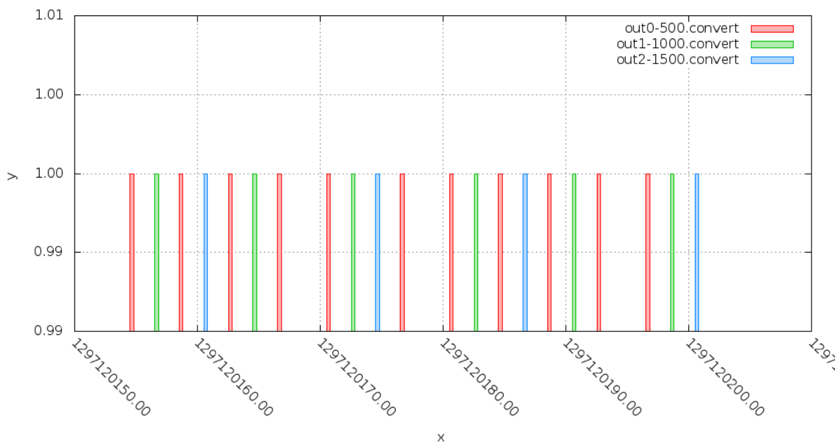


Figure 24: Testing `DRRSched` configuration: distribution of packet at output over time

5.4 Weighted Round Robin Scheduler

We implement this scheduler in two versions. The first is compound element 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. Figure 25 is an example of WRR with two weights: 1 and 2. Source `s0` has weight 1 and connect to port 1 of `rrsched`. Source `s1`, weight 2, is provided two input ports (0 and 2) of `rrsched`. In practice, we need to take care of distribution of ports to have the fairness in response time. Besides, this approach also loses the sequence of packets in each flow. To test this idea, we build another WRR scheduler supporting three input flows with weight 1, 2 and 3 respectively. The result is shown in figure 26 which presents the timestamp of packets at output link.

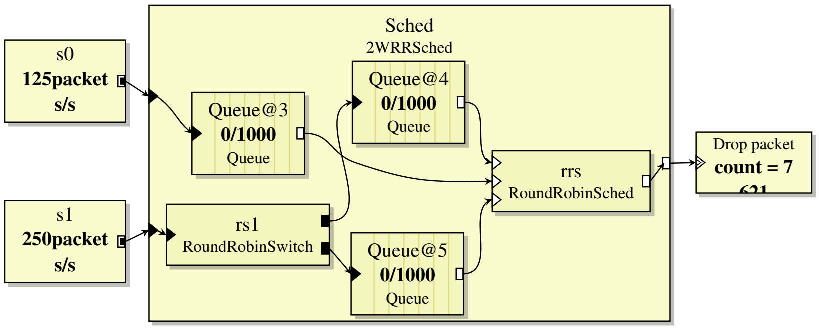


Figure 25: Configuration of `WRRSched` compound element with two input flows and weight 1, 2

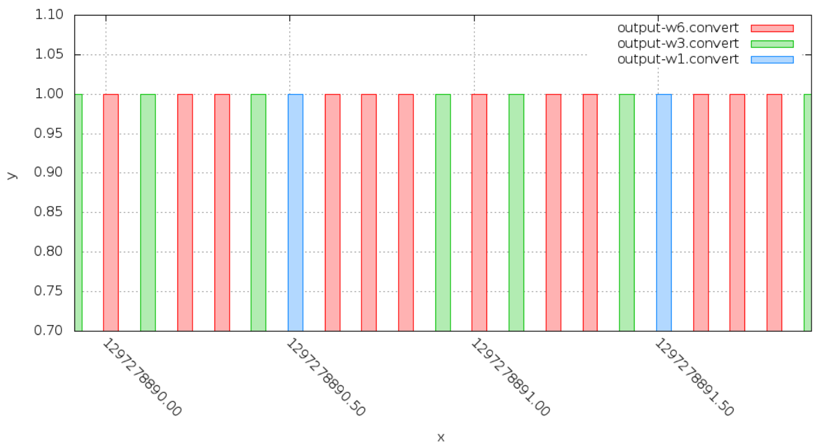


Figure 26: Testing `WRRSched` compound element: distribution of packet at output over time

5.4.2 WRR scheduler - new element

Location:
`elements/wrrsched.{cc,hh}`
`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. Another good point of this element, compared with the above compound element, is that it guarantees the sequence of packets in each flow. 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 (virtual input ports), 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. WRR in our implementation can be seen as the RR scheduling on virtual ports. We test this new element of WRR with three flows, weights are 1, 2 and 3. The result is shown in figure 27.

Figure 27: Testing `WRRSched` element: distribution of packet at output over time

5.5 Weighted Deficit Round Robin scheduler

Similar to Weighted Round Robin (WRR), we also have two versions for Weighted Deficit Round Robin scheduler: compound element and new element.

5.5.1 WDRR scheduler - compound element

Location: `4-scheduler/WDRR_Sched.click`.
The way of implementation is the same as WRR scheduler (compound element), that is to provide more input ports of Deficit Round Robin scheduler to the high weight input flows. Figure 28 shows the distribution of packets of each flow on the output link. We setup three flows with packet length 500, 1000, 1500 byte, and weights 1, 2, 3 respectively. We can see that a number of packets of each flows is nearly the same as we expect.

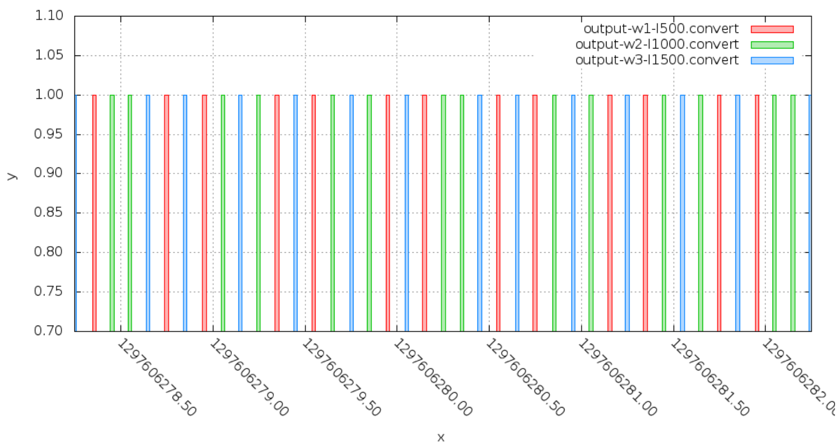


Figure 28: Testing `WDRRSched` compound element: distribution of packet at output over time

5.5.2 WDRR scheduler - new element

Location:
`elements/wdrr.{cc,hh}`
`4-scheduler/WDRR_Sched_element.click`
There are some problems that motivate us to implement a new element for WDRR:

- Quantum is fixed. It is not fair to flows having large size packets.
- Weights cannot be positive real number.
- Weights cannot be changed in runtime.

By inheriting from source code of DRR scheduler, we have just needed to modify at the time updating deficit:

$$new\_deficit = old\_deficit + quantum * weight;$$

We test this element with three flows. Flow0 has 100 byte of packet length, weight 0.2. Flow1 has 200 byte of packet length, weight 0.4. Flow2 has 400 byte of packet length, weight 0.8. We use small weights because packet length in these flows is smaller than quantum value (500) and we do not want burst duration on the output link. Output flow is shown in Figure 29. We can see that with these parameters, WDRR scheduler operates like a Round Robin scheduler.

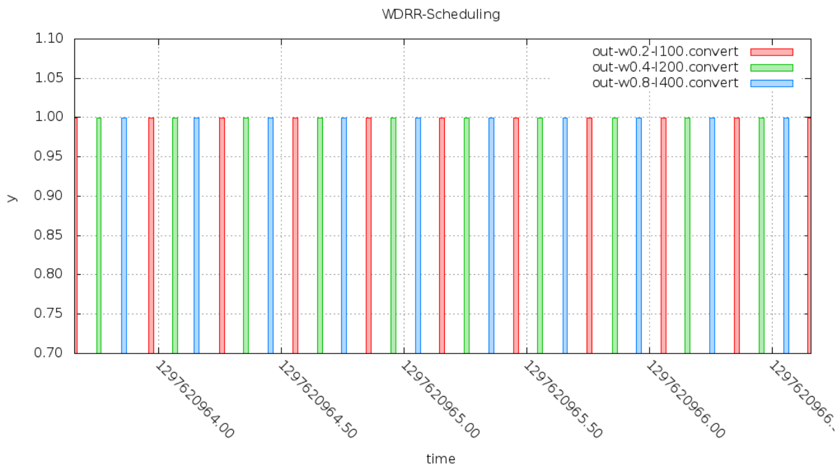


Figure 29: Testing `WDRRSched` element: distribution of packet at output over time

Since this element allows real numeric weight, there are some applications of this element:

- Round Robin scheduling (figure 29). For each flow, setup

$$weight = \frac{avg\_packet\_length}{quantum}$$

- Round Robin scheduling with some burst duration. For each flow, setup

$$weight = \frac{burst\_duration * avg\_packet\_length}{quantum}$$

- Give more fairness in response time. For example, assume there are two flows: packet length in flow 0 is 100 byte, and packet length in flow 1 is 200 byte. With the built-in DRR scheduler, the sequence of packets on the output link is:

$$f0, f0, f0, f0, f0, f1, f1, f0, f0, f0, f0, f0, f1, f1, f1, ...$$

where $f0$, $f1$ are represented to a packet from flow0 and flow1 respectively. We can do more better by using WDRR scheduler with weights 0.2, 0.2. The sequence of packets is changed as:

$$f0, f0, f1, f0, f0, f1, f0, f0, f1, f0, f0, f1, f0, f0, f1, ...$$

5.6 `SetVirtualClock` element

Location: `elements/setvirtualclock.{cc,hh}`
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 values (last virtual time, last real time and tag). Each time a packet arrives, it will set a new tag into timestamp annotation of packet. Computation of tag and virtual time is based on these equations:

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

To use this element, user needs to provide: *rate* (bandwidth of a flow), *max_bw* (maximum bandwidth of output link) and *current_bw* (current used bandwidth of output link). At runtime, this element allows user to read *last_tag* and also modify *rate*, *max_bw*, *current_bw*. When a packet arrives to this element, it does the following tasks:

- Update virtual time:

$$last\_virtual\_time+ = \frac{(current\_real\_time - last\_real\_time) * max\_bw}{current\_bw}$$

- Update tag:

$$last\_tag = \max(last\_tag, last\_virtual\_time) + \frac{packet\_length}{rate}$$

- Update packet's timestamp annotation to *last_tag*
- Update parameter of `SetVirtualClock`: *last_real_time* to the current time.

Note: when configuring `SetVirtualClock` with $max\_bw = current\_bw$, the virtual time is exactly the real time. We can use this property to implement GCRA and Virtual Clock scheduling.

5.7 Virtual Clock scheduler

Location: `4-scheduler/VC_Sched.click`
We implement the Virtual clock scheduler with support of `SetVirtualClock`. Each flow is connected to a `SetVirtualClock` and then to a `TimeSortedSched` (Figure 30). Element `SetVirtualClock` is set up with $MAXBW = CURRENTBW$. With this configuration, packets out of `SetVirtualClock` will have a new timestamp following the rule of Virtual Clock scheduling. At that time, packets will go through element `TimeSortedSched` which is a scheduler based on timestamp of packets.

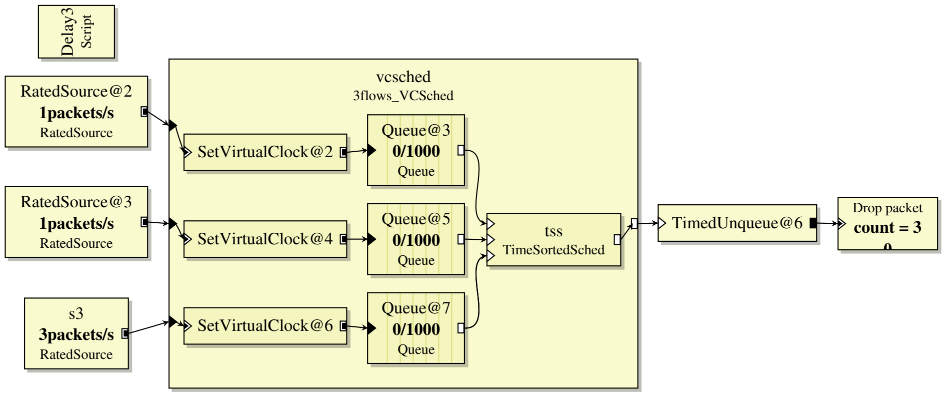


Figure 30: `VirtualClock` scheduler with 3 flows

Scenario of testing Virtual Clock scheduling: There are three flows,

- Flow 0: Rated source, packet length 1 byte, rate 1 pps.
- Flow 1: Rated source, packet length 1 byte, rate 1 pps.
- Flow 2: Rated source, packet length 1 byte, rate 3 pps.

We set up three `SetVirtualClock` with the same parameters:

$$RATE = 1, MAXBW = 3, CURRENTBW = 3$$

These parameters said that each flow can only use one-third bandwidth of output link. Output link speed is 1pps. Flow 2 is postponed 5 second later than flow 0 and flow 1. The result is shown in figure 31. We can see that when flow 2 starts to send packets, it does not make large effect to other flows but shares the output with others.

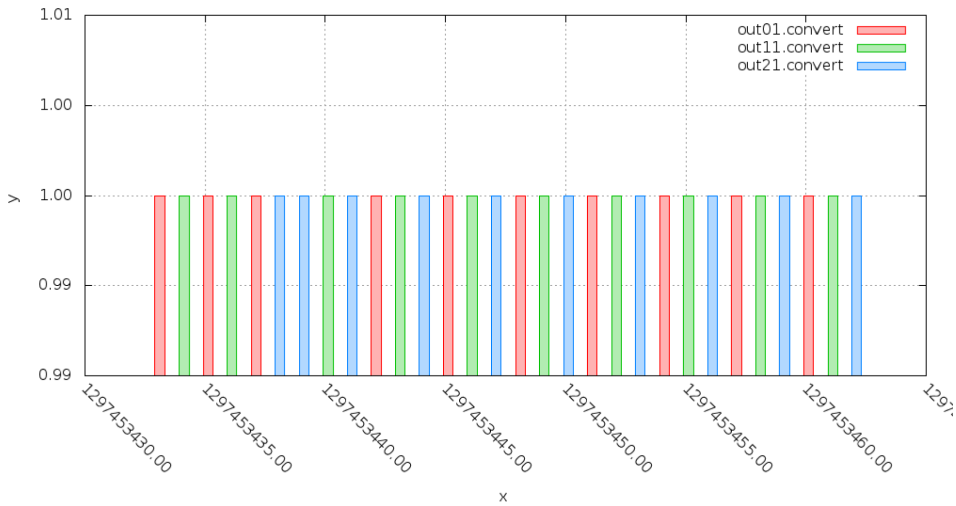


Figure 31: Testing `VirtualClock` scheduler with 3 flows

5.8 Weighted Fair Queueing (WFQ) scheduler

Location: `4-scheduler/WFQ_Sched.click`
Figure 32 shows the Click configuration of WFQ with three input flows. We also use a `SetVirtualClock` for each flows before letting them go through the `TimeSortedSched`. The problem in WFQ scheduling is that we need to take care of input flows to know which one is active. We use `UpdateCurrentBW` and `Counters` to do this job. Assume that packet rate of input flows is larger than 1 (at least 1 pps). `Counter` is used to know whether the appropriate input flow is active or not in the last second. These `Counters` are reset to 0 each second. Based on information from `Counter`, script `UpdateCurrentBW` will adjust parameter `currentbw` of each element `SetVirtualClock` to the sum of bandwidth of current active flows.

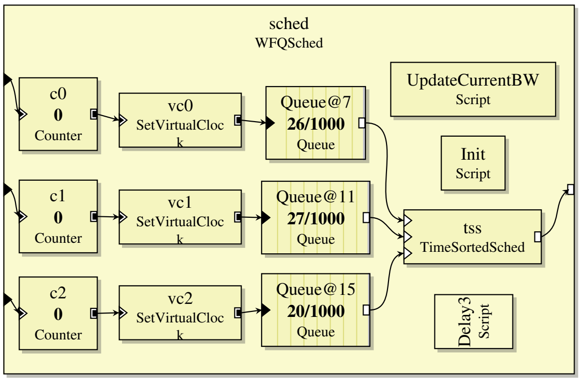


Figure 32: `Weighted Fair Queueing` scheduler with 3 flows

Testing scenario: three flows have the same packet rate 1 pps, and packet length 3 byte. Flow 0 and flow 1 are started at the same time, flow 2 is started 15 seconds later. Output packet rate is 1 pps. Figure 33 shows the output packets whose color is representation of flows. We see that flow 0 and flow 1 are not lost bandwidth when flow 2 starts.

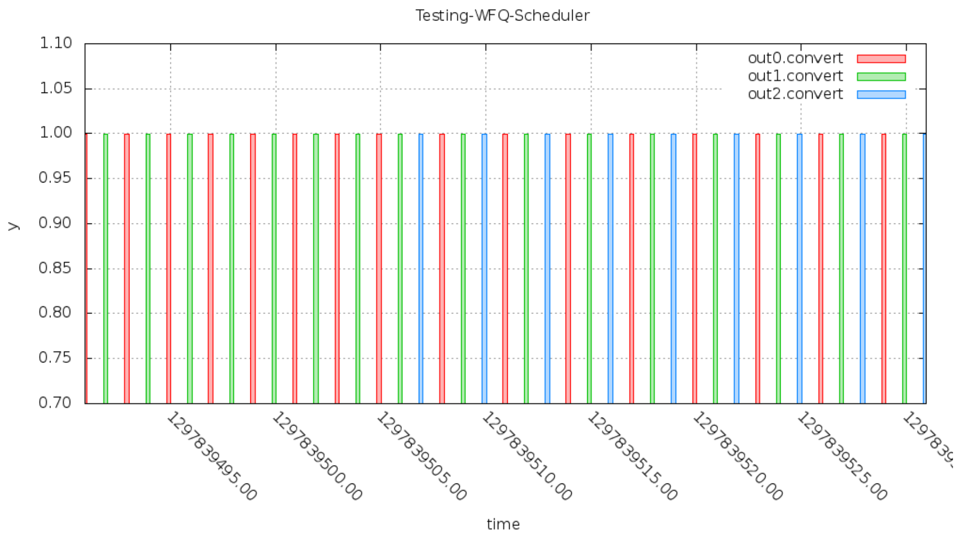


Figure 33: Testing `Weighted Fair Queueing` scheduler with 3 flows

We use the same scenario to Virtual Clock scheduler (section 5.7), and result is shown in figure 34. Since Virtual Clock scheduler always assumes that all flows are active, it is not good when a flow is started long time after other flows.

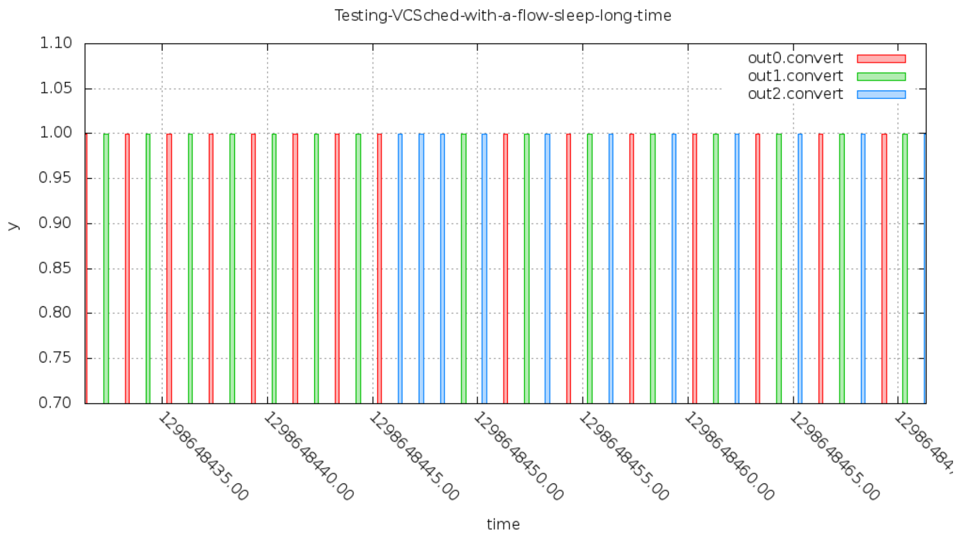


Figure 34: Testing `Virtual Clock` scheduler with the same scenario of WFQ scheduler

5.9 Earliest Due Date(EDD) scheduler

Location:
`4-scheduler/EDD_Sched.click`
`elements/eddsched.{hh,cc}`
`elements/eddqueue.{hh,cc}`
`elements/shiftqueue.{hh,cc}`
To implement EDD scheduler, there are two interesting problems:

- **Assigning deadline**. There are two algorithms: Delay EDD and Jitter EDD. Since time is limited, we do not implement this function.
- **Deadline guarantee**. Beside implementing the basic function of EDD scheduler that the smallest deadline packet goes first and drop expired deadline packet at output, we see that if we can estimate the time that a packet is chosen by EDD scheduler before puting it into waiting queue, we can do better for the future packets. Although queue still has some empty space, but we know that a new packet will be dropped at the time chosen by scheduler, we should drop it right now and hold an empty space for others.

Assume that all flows of packet come to EDD scheduler with assigned deadlines. We consider two ways of storing deadline value: in timestamp annotation or in payload of packet. The first way may be suitable to Delay EDD and Jitter EDD. The second way can be applied to all cases generally. Some additional elements are implemented to do store deadline and do EDD algorithm:

- Element `SetTimestamp2`: it is inheritted from element `SetTimestamp` but user can use additional keyword `ADD` to define the expected deadline by changing timestamp annotation of packet. This element is also used to store deadline in payload of packet by combining it and element `StoreTimestamp`. See figure 35.

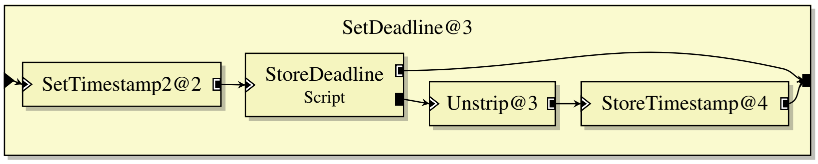


Figure 35: Element `SetTimestamp2` and `StoreTimestamp` are used to store deadline in Timestamp annotation or payload

- Element `CheckTimestamp`: this element drops packets if it is late (its timestamp annotation is in the past).
- Element `EDDSched`: EDD scheduler. This scheduler only does the basic function: choose next packet with the minimum deadline. If packet is late, it is killed or dropped. This element can check packet's deadline which is in the timestamp annotation or in the head of payload (the first eight bytes).
- Element `EDDQueue`: This queue is `ThreadSafeQueue` with additional function: A packet is pushed successfully to this queue only if

$$(Packet\_Deadline - Current\_time) * Average\_output\_rate > Current\_queue\_length$$

- Element `ShiftQueue`: This queue is `SimpleQueue` with additional function: If queue is full, packets in queue will be remove if they could be expired in future. More detail is in section 5.9.3.

5.9.1 EDD scheduler with basic functions

After assigning deadline to packet, the EDD scheduler operates similar to a FIFO scheduler. For that reason, EDD scheduler is a "FIFO scheduler" with dropping expired packets. Figure 36 shows two configurations of EDD scheduler with one thing difference: in the right figure, by using element `EDDSched`, if packet misses its deadline, the scheduler will drop it and find another packets immediately (no need to run another round). If EDD scheduler is implemented by compound element (`_edds0`), expired packet is checked in `CheckTimestamp` after the scheduler chooses it, so it needs to request the scheduler again (one more round) to choose another one. The performance of `_edds1` should be better than `_edds0` in the cases of many expired packets.

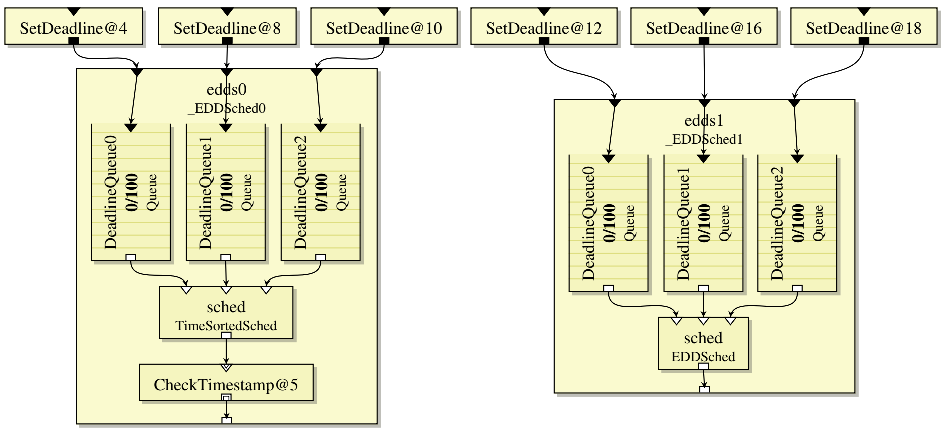
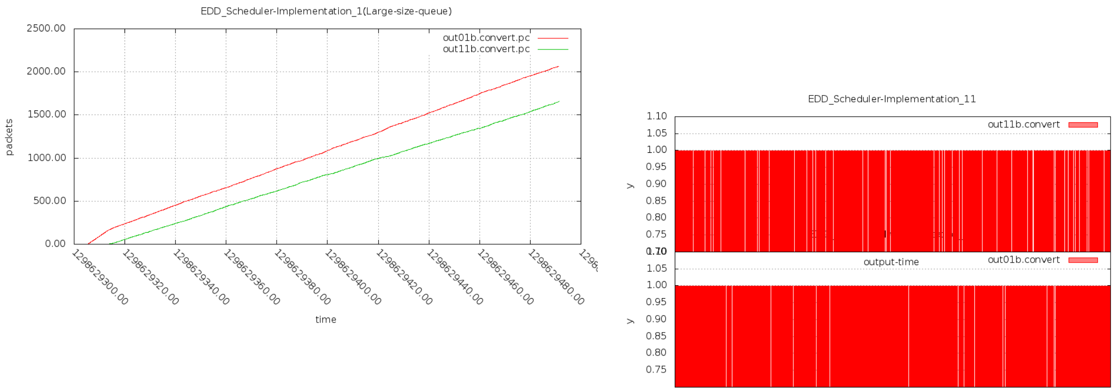


Figure 36: Two configurations of EDD Scheduler with basic function

Scenario of testing: there are two flows `RatedSource` with the same rate (100 pps). The deadline of the first flow is 6 seconds, and deadline of the second flow is 8 second. The output rate is 20 pps. These



(a) Number of packets over time

(b) Distribution of packets of each flow at output over time

Figure 37: Testing results when using large size queues with EDDSched compound element `_EDDSched1`: flow 0 is represented by `out01b`, flow 1 is represented by `out11b`

configured parameters are based on these constraints:

$$d_1 > d_0 \tag{1}$$

$$d_0 > \frac{R_i}{R_o} \tag{2}$$

$$d_1 < \frac{(d_1 - d_0) * R_i}{R_o} \tag{3}$$

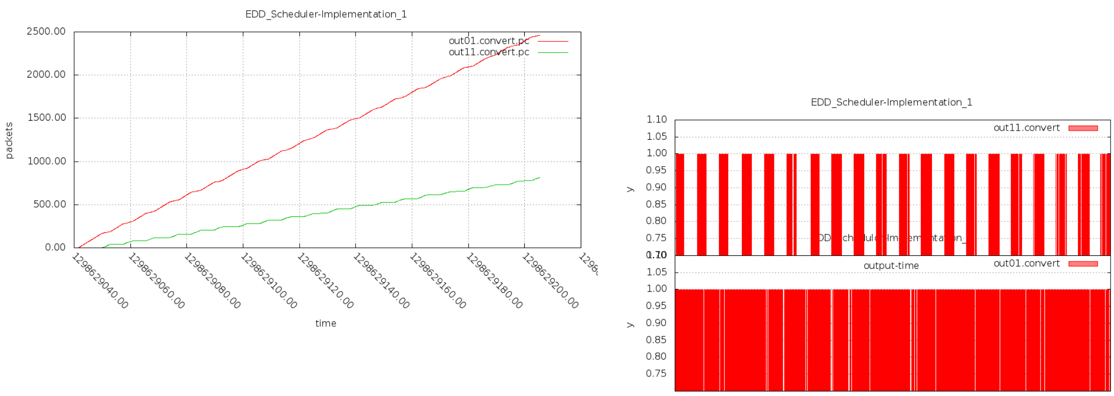
$$d_1 < \frac{d_0 * R_i}{R_o} \tag{4}$$

where d_i is the deadline of flow i, R_i is the input rate, R_o is the output rate. We try to create a scenario in which one flow dominates others which is described in these above inequations. At the beginning, we setup a large capacity of the queues storing packets from these flows. We observe that this capacity is about $FLOW\_RATE * DEADLINE \rightarrow 100 * 8 = 800$. Value $DEADLINE$ is the maximum time a packet staying in queue before being processed. The result is shown in figure 37.

We do another test with one thing changed: queue size for each flow is reduced to $OUTPUT\_RATE * DEADLINE * 2 + 5$. The reason is that the time of scheduling all packets in a queue should be smaller than or equal to the deadline value. Queue size for the first flow is 245 and for the second flow is 325. The result of this test is shown in figure 38. We see that, when queue size is reduced, the EDD scheduler treat not fair to large deadline flow. The reasons could be:

- New packets are refused because of full queue while some packets in queue are too old and will be killed when scheduled.
- Life time (deadline) of packets is damaged by small deadline flow.

The second reason is difficult to deal with because it is from EDD scheduler. We try to improve EDD scheduler to get back some bandwidth from flow 0 to flow 1 by dealing with the first reason. The idea of solution is to remove packets that we know or be sure that they will be expired. The next two subsections will try to improve the fairness in the case of limitted size queue so that it could operate as well as the case of large size queue (figure 37). Assume that we know the output packet rate, input packet rate, and there is only two input flows.



(a) Number of packets over time

(b) Distribution of packets of each flow at output over time

Figure 38: Testing results when using limitted-size queues with EDDSched compound element `_EDDSched1`: flow 0 is represented by `out01`, flow 1 is represented by `out11`

5.9.2 Improving EDD scheduler with dropping policy at inputs

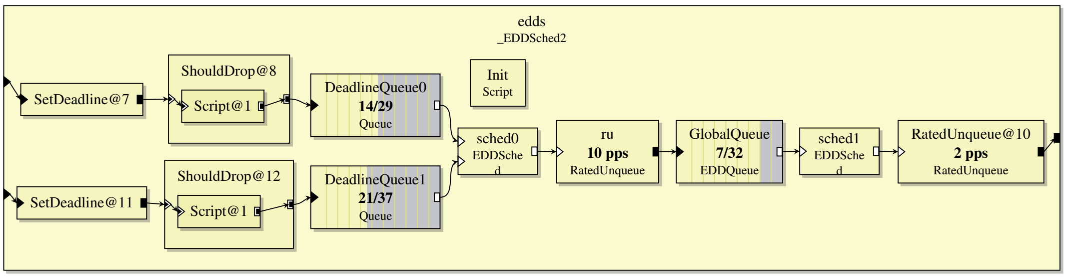


Figure 39: EDD Scheduler with improvement `_EDDSched2`

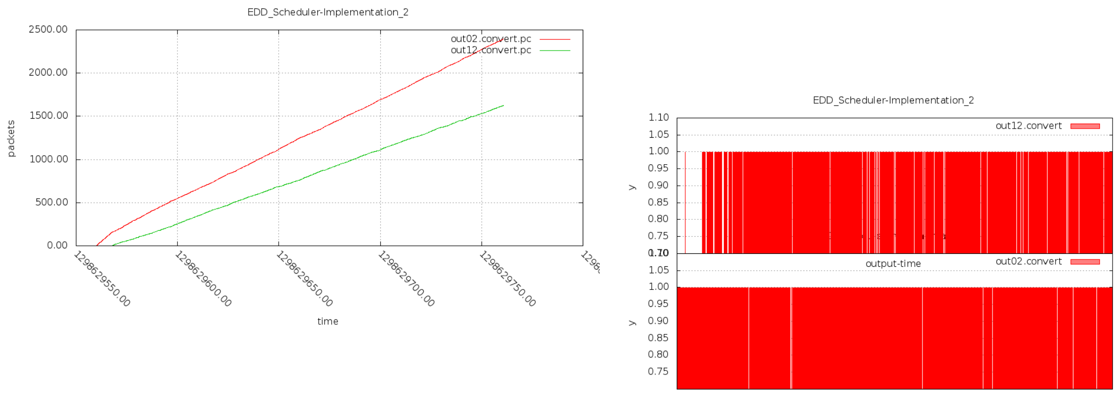
Figure 39 shows the Click configuration of improved EDD scheduler with this dropping policy that does not always allow packets stored in queue (`DeadlineQueue`). Element `ShouldDrop` makes the decision of dropping packets: packet is dropped if

$$\alpha * OUTPUT\_RATE * DEADLINE < GlobalQueue.length + DeadlineQueue.length$$

Parameter α can be configured by user. Normally, we set $\alpha = 1.7$. Since some packets in `GlobalQueue` can be expired, we use parameter α to control this event. `GlobalQueue`, an instance of element `EDDQueue`, denies all packets that will be expired in the future if:

$$OUTPUT\_RATE * (PACKET\_DEADLINE - CURRENT\_TIME) < EDDQUEUE\_LENGTH$$

We test this configuration with the same parameters as in the second test in section 5.9.1. In figure 40, the result shows that flow 1 has more chance to send out packets with this improvement. Note that queue size is limited.



(a) Number of packets over time

(b) Distribution of packets of each flow at output over time

Figure 40: Testing results when improving EDD scheduler with dropping policy: flow 0 is represented by `out02`, flow 1 is represented by `out12`

5.9.3 Improving EDD scheduler with `ShiftQueue`

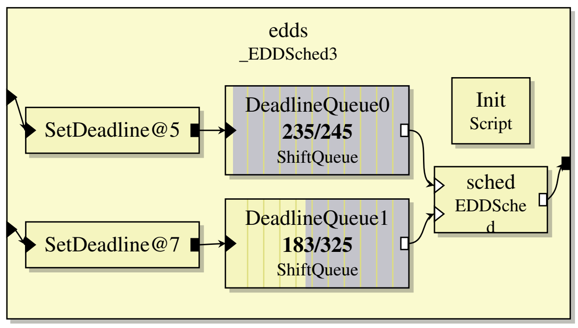


Figure 41: EDD Scheduler with improvement `_EDDSched3`

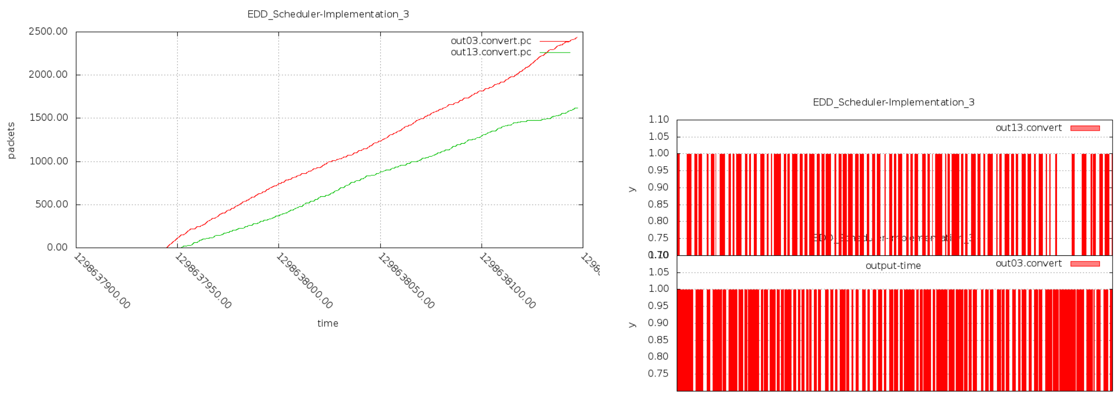
When observing operation of EDD scheduler, we see that in the high deadline queue, sometimes most packets are expired or will be expired when scheduled. But at that time, queue is full, new packets which could be scheduled, can not enter to it and be dropped. When this queue is scheduled, all expired packets are removed and new packets comes but with high deadline and continue to wait for their turn. We have rebuilt the `SimpleQueue` to `ShiftQueue` to avoid this phenomenon by allowing to remove old packets when queue is full. We do not check residual time of packets everytime because of performance. When a packet is pushed to `ShiftQueue`:

- If queue is not full, packet is stored in queue normally.
- If queue is full: remove packet by packet from tail of queue if

  $$OUTPUT\_RATE * (PACKET\_DEADLINE - CURRENT\_TIME) < EDDQUEUE\_LENGTH$$

- After cleaning queue and having some free space, new packet is added to queue. Otherwise, it is dropped.

Figure 41 is the configuration using this `ShiftQueue`. We also use the same second scenario in the section 5.9.1 to test this configuration. In figure 42, we see that the fairness is comparable to the case of large size queue.



(a) Number of packets over time

(b) Distribution of packets of each flow at output over time

Figure 42: Testing results when improving EDD Scheduling with `ShiftQueue`: flow 0 is represented by `out03`, flow 1 is represented by `out13`

6 Congestion control

6.1 Weighted RED buffer management

We want to build a Weighted RED (WRED) based on RED element. A WRED allows us to define some ranges of bandwidth, each range has a particular probability of dropping packets. The idea of implementation: Packets are classified by their rate (or a range of rate), and then let them go through a RED element used for this rate. We check this idea by writing a WRED that supports two ranges of rate, see figure 43.

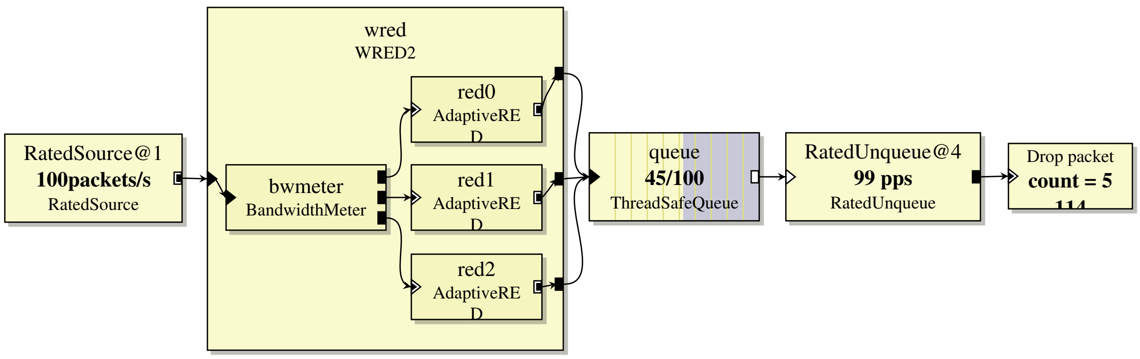


Figure 43: One simple implementation of `WRED` element

To test new element of WRED, we calculate probability of dropping packets by fraction of number of dropped packets over number of packets in a particular range of rate. Since RED element accumulates dropped packets over time, we have to use another counter to count number of dropped packets (be able to reset this counter for another average queue size). When observing the operations of RED element, we recognize that with a particular output link speed, the average queue size goes to a stable value. From that, our testing strategy is to estimate probability of dropped packets as following:

- Output link speed begins with input link speed.
- Reset all counters: number of input packets, number of dropped packets
- Restart input source, wait for stable average queue size (we let it wait for 10 second).
- Stop source, and then calculate probability of dropped packets.

- Print to log file information of this probability and average queue size (getting from element `RED`).
- Change output link speed to another value and return to step 2.

For testing, we set up two RED elements with parameters:

- RED 1: $MIN\_THRESH = 40, MAX\_THRESH = 80, MAX\_P = 0.4$ for a range of input bandwidth $0.2KBps \rightarrow 12KBps$.
- RED 2: $MIN\_THRESH = 50, MAX\_THRESH = 80, MAX\_P = 0.3$ for a range of input bandwidth $12KBps \rightarrow \infty$.

Testing result is shown in figure 44 (lines are smooth using `sbezier`).

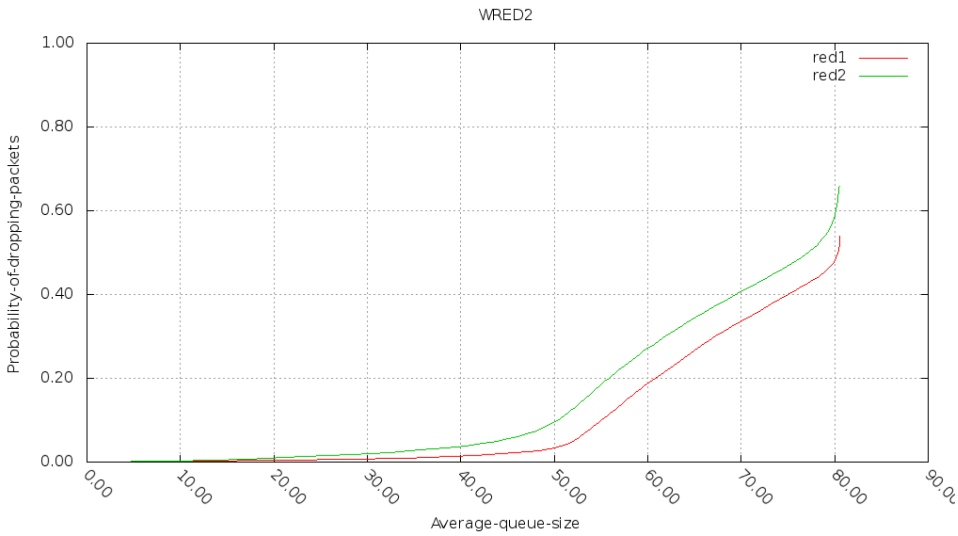


Figure 44: Testing `WRED2`

A Appendix

A.1 Command init.sh

```
DESCRIPTION
  Initialize enviroment for ClickLabs: Setting path for visual-clicky.sh
  (tool for visualize click experiment) and update new implemented click elements.

  SYNOPSIS
    init.sh  [-h | --help]
             [-s | --click-src CLICK-SOURCE-PATH]
```

A.2 Command visual-clicky.sh

```
DESCRIPTION
  Shell script to visualize click experiment using clicky

  SYNOPSIS
    visual_clicky.sh  [-f | --file] CLICK_PATH_FILE
                      [-p | --port PORT]
                      [-s | --ccss CCSS_FILE]
                      [-h]
```

A.3 Command update-elements.sh

```
 DESCRIPTION
   Support adding new elements into Click source file and then compile them as Click element

 SYNOPSIS
   update-elements.sh

        -h
            Show information about update-elements.sh
```

A.4 Command eclick-compile.sh

```
 DESCRIPTION
   Extend the ability of click file to support multiple included clicks into file

 SYNOPSIS
   eclick-compile.sh

        -f ARG
            Extended Click file

         ARG
            Extended Click file

```

```
        --file ARG
            Extended Click file

        -h
            Help information

        --help
            Help information

        -o ARG
            Output file (click file). Default is /dev/stdout

        --ouput ARG
            Output file (click file). Default is /dev/stdout
```

A.5 Command convert-click-dump.sh

```
DESCRIPTION
  Transform tcpdump-like packet trace file into human-readable file

SYNOPSIS
  convert-click-dump.sh

        --help
            Help on convert-click-dump.sh

        -h
            Help on convert-click-dump.sh

        --dumpfile ARG
            Dump file from click or tcpdump

        -f ARG
            Dump file from click or tcpdump

         ARG
            Dump file from click or tcpdump

        --packet-count
            Insert more parameter: packet count

        -c ARG
            Exit after get enough number of packet (ARG)

        -o ARG
            Output file when conversion is finished.
```

A.6 Command draw-graph.sh

```
DESCRIPTION
  Drawing graph from traced packet dump file

SYNOPSIS
```

```
    draw-graph.sh

        --help
            Help on draw-graph.sh

        -h
            Help on draw-graph.sh

        -o ARG
            Output file name (for example: h.png)

        -f ARG
            Dump file from click or tcpdump. Can use this option many
                times to draw some lines in graph

        --packet-count
            Insert more parameter: packet count

        --xrange ARG
            Relative range of X axes, based time is the beginning
               time (for example: 0:1, or 10:100, ...)

        --xlabel ARG
            Label of X axes.

        --xcol ARG
            Column of X in dataset (dumpfile)

        --yrange ARG
            Range of Y axes (for example: 0:1, or 10:100, ...)

        --ylabel ARG
            Label of Y axes.

        --ycol ARG
            Column of Y in dataset (dumpfile)

        --title ARG
            Title of the graph. For example: Arrival Curve of input
               packet

        --plot-type ARG
            Type of plotting data. It can be: RATE, COUNT (Default)
               or DENSITY

        --data ARG
            Data for plotting

        --multiplot
            Plots place separately and vertically

        --template ARG
            Plot template

        --plot-option ARG
            Plot option
```

A.7 Command draw-graph-negotiation.sh

```
  DESCRIPTION
    Drawing graph from traced packet dump file, especially for frame
       relay network

  SYNOPSIS
    draw-graph-negotiation.sh

         --help
             Help on draw-graph-negotiation.sh

         -h
             Help on draw-graph-negotiation.sh

         -o ARG
             Output file name (for example: h.png)

         -f ARG
             Dump file from click or tcpdump. Can use this option many
                 times to draw some lines in graph

         --data ARG
             Data for plotting

         --cir ARG
             Committed Information Rate (packet per second)

         --cbs ARG
             Committed Burst Size (packets)

         --ebs ARG
             Excess Burst Size (packets)

         --access-rate ARG
             Access Rate (packet per second)

         --base ARG
             Base time, from this we plot
```