

OMNeT++ Fuzzy IP Scheduler

Overview, implementation, and analysis

Tomas-Adrian Boboi

Universitatea Politehnica Timișoara

2021
March

Contents

1 Initial requirements	3
2 Network structure	4
2.1 IpScheduler	4
2.2 PacketHandler	5
2.3 User	5
2.4 Queue	5
2.5 Sink	6
2.6 Scheduler	6
3 Scheduler variations	6
3.1 Priority Queueing Scheduler	6
3.2 Round Robin Scheduler	7
3.3 Weighted Round Robin Scheduler	8
4 Comparative analysis	9
4.1 Queue Lengths	10
4.2 Packet Lifetime	11
5 Conclusions	12
6 References	14

1 Initial requirements

The simulation model

The simulation model consists of the following OMNeT++ modules:

1. A number of mobile users. In the first stages of the model you can implement two identical users, then you can consider a number of K users, organized as an array of users. Each user generates IP packets according to a certain pattern: e.g. an IP packet at certain (random) time intervals, or a user generates files, a file consisting on a (random) number of IP packets. At the beginning the IP packets can be considered of fixed length, i.e. one IP packet = 1500 bytes. A user has a certain priority. There can be for example 3 or 4 priority levels. For 3 priority levels, they are (in increasing order): LP (low priority), medium priority (MP) and HP (high priority). For 4 levels, they are: non real-time (nrt) low priority and nrt HP, real-time (rt) LP and rt HP.
2. A scheduler. The scheduler is situated in the same Omnet module as the queues. The queues are not per user, but per priority class. This means that the packets arriving from LP user will be stored in the LP queue, the packets from MP and respectively HP users will be stored in the MP and respectively HP queue. The scheduler reads the lengths of the queues and implements a scheduling algorithm that determines which queue will send data. The sending of a packet takes a time equal to its length divided by the line rate (i.e. 1500 bytes / 1 Mega bit per second). The scheduler cannot send another packet during this time interval.
3. A sink. The sink models the destination of the data. When the data (i.e. IP) packets created by an user arrive to the sink module, the sink simply deletes the OMNeT++ messages representing the data packets. Also, the sink is used to collect statistics about the simulation, statistics that can be for each user, for each group of users (e.g. Low, Medium and High priority users) and/or for the entire system. These statistical information can be: the number of data packets that arrive to the sink, the mean, minimum and maximum delay of the data packets, etc.

In the OMNeT directories there is one called "samples", with different simulation models implemented in OMNeT++. From these samples, you can use as a starting point for your model the sources from the "fifo" system.

Basic/minimal requirements

Implement the simulation model described above. The scheduling algorithm is not very important in this stage, it can be a simple round robin: each nonempty queue sends a data packet. Or it can be a priority queueing algorithm: a queue is not served until there are non-empty higher priority queues.

For exam, there are two alternatives to improve the project

Implement one of the following scheduling algorithms:

- Priority queueing (a lower priority queue is served if and only if all higher priority queues are empty)

- A weighted round robin: an action takes place every time when there are resources available. The winner of the action will be served. The criteria for the action is, firstly, the time elapsed since the user was served last time. Then, we can improve the algorithm by introducing weights (positive numbers ω_1). The time elapsed since the user was served last time is multiplied with user's weight. Then, a user with a higher weight will be served more often. Compare the performance (i.e. average delay, minimum and maximum delay) of the implemented algorithms.

2 Network structure

The network is described by the `IpScheduler.ned` file, which contains the users, which generate the IP packets, the packet handler, which handles the packets received from the users, and the sink, which receives the packets sent from the packet handler.

2.1 IpScheduler

`IpScheduler` is the main network, where all the sub-components reside.

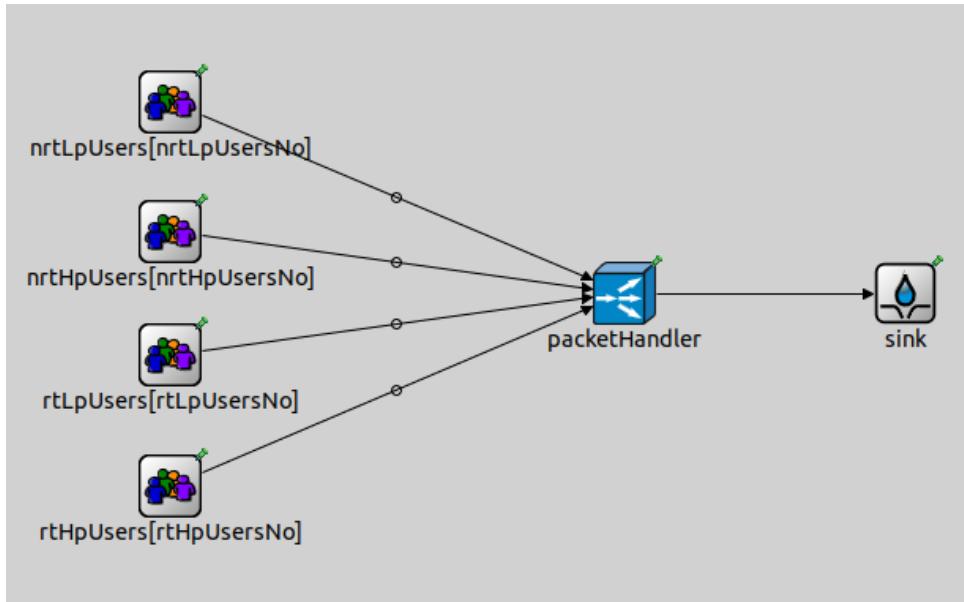


Figure 1: The main network structure

2.2 PacketHandler

PacketHandler is the component responsible with receiving the packets from the users, storing them in the corresponding queue, depending on the priority level of the users, scheduling the sending of the packets, and sending the packets to the sink.

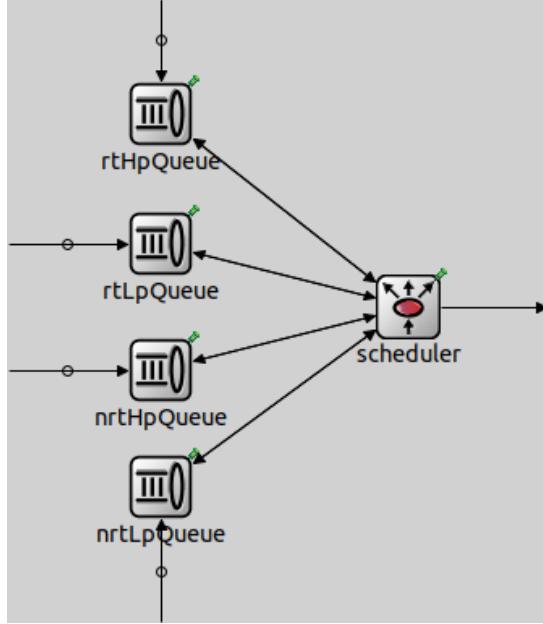


Figure 2: The packet handler structure

2.3 User

The **User** is the component which generates IP packets at a certain rate, and sends them to be handled by the **PacketHandler**. The users are grouped in priority classes:

- non-real-time, low priority (nrtLp)
- non-real-time, low priority (nrtHp)
- real-time, high priority (rtLp)
- real-time, low priority (rtHp)

The size of the generated IP packets, the packet generation rate, and the number of users in each priority class can be configured in the `ipsched/config/users.ini` and `ipsched/config/user_packets.ini` files.

2.4 Queue

The **Queue** is the module which stores the IP packets generated by the users, and sends them towards the sink, whenever the scheduler allows it. There are four queues, one for each priority class, and they all communicate with the scheduler bidirectionally: they receive control messages from the scheduler, and send packets to the scheduler, when asked to. The *Queue* class contains a `cPacketQueue` attribute, where the IP packets are stored. The **Queue** module is also responsible with logging each queue's length, so we can collect data about the length of each queue at any given time.

2.5 Sink

The **Sink** is a module whose only purpose is to receive the IP packets, delete them, and record statistics about them (packet lifetime, total number of packets, etc.).

2.6 Scheduler

The **Scheduler** can be considered the most complex module of the system, and the most important, because it is responsible for deciding which queue sends a packet towards the sink. This job can be accomplished in multiple ways, and the current project provides three scheduler implementations: Priority Queueing, Round Robin, and Weighted Round Robin.

3 Scheduler variations

The current project provides three implementations for the scheduler, each with its particularities and different levels of efficiency.

3.1 Priority Queueing Scheduler



The screenshot shows a terminal window with a dark background and three colored status indicators (red, yellow, green) at the top. The terminal displays a block of C++ code. The code is a switch statement that checks the length of four different queues: rtHpQueue, rtLpQueue, nrtHpQueue, and nrtLpQueue. If any queue has a non-zero length, it sends a 'schedulerMessage' to a specific port and sets a flag 'sent' to true. Finally, it schedules a message to be sent at a future time, adding the current simulation time plus a parameterized packet generation delay, truncated to microseconds, and readyToScheduleMessage.

```
1 if (getQueueLength("rtHpQueue") > 0)
2 {
3     send(new cMessage("schedulerMessage"), "rtHpQueueControl_out");
4     sent = true;
5 }
6 else if (getQueueLength("rtLpQueue") > 0)
7 {
8     send(new cMessage("schedulerMessage"), "rtLpQueueControl_out");
9     sent = true;
10 }
11 else if (getQueueLength("nrtHpQueue") > 0)
12 {
13     send(new cMessage("schedulerMessage"), "nrtHpQueueControl_out");
14     sent = true;
15 }
16 else if (getQueueLength("nrtLpQueue") > 0)
17 {
18     send(new cMessage("schedulerMessage"), "nrtLpQueueControl_out");
19     sent = true;
20 }
21
22 if (!sent)
23 {
24     scheduleAt(
25         simTime() + SimTime(par("packetGenerationDelay").doubleValue(),
26                               SIMTIME_US)
27             .trunc(SIMTIME_US),
28         readyToScheduleMessage);
29 }
```

Figure 3: The inner workings of the Priority Queueing scheduler

The priority queueing algorithm, according to the requirements, allows a queue to send a packet if and only if the higher priority queues are empty.

As we can see in the code, we first check the length of the queue with the highest priority, then, if it is zero, we move on to check the length of the queue below it priority-wise, and so on. If the scheduler decides that a queue should send a packet, it sends a control message to the respective queue, telling it to send a packet. The queue will then receive that message, and know that it came from the scheduler, and sends a packet.

If no packet was requested, that means that all the queues' lengths are zero, and the scheduler enters a sleep state, which is equal to the average time needed for a packet to arrive in any one of the queues.

3.2 Round Robin Scheduler

```

1 simtime_t times[4] = {lastServed_nrtLp, lastServed_nrtHp,
2                         lastServed_rtLp, lastServed_rtHp};
3 sortTimes(times);
4 bool sent = false;
5
6 for(int i = 0; i < 4 && !sent; i++)
7 {
8     if (times[i] == lastServed_nrtLp)
9     {
10         if (getQueueLength("nrtLpQueue") > 0)
11         {
12             send(new cMessage("schedulerMessage"), "nrtLpQueueControl_out");
13             sent = true;
14         }
15         lastServed_nrtLp = simTime();
16     }
17     else if (times[i] == lastServed_nrtHp)
18     {
19         if (getQueueLength("nrtHpQueue") > 0)
20         {
21             send(new cMessage("schedulerMessage"), "nrtHpQueueControl_out");
22             sent = true;
23         }
24         lastServed_nrtHp = simTime();
25     }
26     else if (times[i] == lastServed_rtLp)
27     {
28         if (getQueueLength("rtLpQueue") > 0)
29         {
30             send(new cMessage("schedulerMessage"), "rtLpQueueControl_out");
31             sent = true;
32         }
33         lastServed_rtLp = simTime();
34     }
35     else if (times[i] == lastServed_rtHp)
36     {
37         if (getQueueLength("rtHpQueue") > 0)
38         {
39             send(new cMessage("schedulerMessage"), "rtHpQueueControl_out");
40             sent = true;
41         }
42         lastServed_rtHp = simTime();
43     }
44 }
45
46 if (!sent)
47 {
48     scheduleAt(
49         simTime() + SimTime(par("packetGenerationDelay").doubleValue(),
50                             SIMTIME_US)
51         .trunc(SIMTIME_US),
52     readyToScheduleMessage);
53 }

```

Figure 4: The inner workings of the Round Robin scheduler

The Round Robin algorithm might look more complex than the previous one, but the underlying principle is in actuality quite simple: the queues which sent a packet recently don't send a packet in the current scheduling cycle, and only the one which sent a packet the longest time ago, sends a packet in the current cycle.

To achieve this, we keep track of the last time at which each queue sent the last packet. When a queue sends a packet, this time is updated to the current simulation time.

The scheduling algorithm computes the minimum of the mentioned times (the earliest time corresponds to the minimum time), and then checks which queue corresponds to this minimum time. After selecting the queue, the algorithm also checks the length of that queue, so we don't pop from an empty queue, and get an error. As stated before, after each queue sends a packet, its corresponding last-sent-time is updated.

3.3 Weighted Round Robin Scheduler

The Weighted Round Robin is the most complex of the three presented in this project, but is also the most effective in scheduling the packets to ensure efficiency and speed.

This algorithm required each class of users to have associated a weight, which will tip the balance in favour of the higher priority users, which will be served more often than the lower priority ones.

The users' weights are multiplied with the time elapsed since the corresponding queue sent a packet, and the resulting weighted-time is the criterion by which the sender queue is chosen.

For example, let us consider two priority classes, *low priority* and *high priority*. If the low-priority queue sent a packet 1ms ago, and the high-priority queue sent a packet 500ns ago, with a classic Round Robin algorithm, the low-priority queue should send a packet. With the weights introduced, the 1ms and 500ns times are multiplied with the weights, say 1 for the low-priority queue, and 4 for the high-priority queue. Therefore, the weighted times, which are the basis on which the scheduler decides which queue sends the next packet, are 1ms for the low-priority queue, and 2ms for the high-priority one. Therefore, to the scheduler, it "appears" as if the high-priority queue is the one which sent a packet a long while ago, and the low-priority queue is the one which sent a packet more recently, and so the high-priority queue sends a packet in the current cycle.

With the weights introduced as described, the queue in priority class i sends packets $\frac{w_i}{w_j}$ times as often as the queue in priority class j , where w_i and w_j are queue i 's weight and j 's, respectively.

```

1  bool sent = false;
2
3  simtime_t lastServedWeightedDelta_nrtLp = (simTime() - lastServed_nrtLp)
4      * userWeights[0];
5  simtime_t lastServedWeightedDelta_nrtHp = (simTime() - lastServed_nrtHp)
6      * userWeights[1];
7  simtime_t lastServedWeightedDelta_rtLp = (simTime() - lastServed_rtLp)
8      * userWeights[2];
9  simtime_t lastServedWeightedDelta_rtHp = (simTime() - lastServed_rtHp)
10     * userWeights[3];
11
12 simtime_t times[4] = { lastServedWeightedDelta_nrtLp,
13     lastServedWeightedDelta_nrtHp, lastServedWeightedDelta_rtLp,
14     lastServedWeightedDelta_rtHp };
15 sortTimes(times);
16
17 for(int i = 0; i < 4 && !sent; i++)
18 {
19     if (times[i] == lastServedWeightedDelta_nrtLp) {
20         if (getQueueLength("nrtLpQueue") > 0) {
21             send(new cMessage("schedulerMessage"), "nrtLpQueueControl_out");
22             sent = true;
23         }
24         lastServed_nrtLp = simTime();
25     } else if (times[i] == lastServedWeightedDelta_nrtHp) {
26         if (getQueueLength("nrtHpQueue") > 0) {
27             send(new cMessage("schedulerMessage"), "nrtHpQueueControl_out");
28             sent = true;
29         }
30         lastServed_nrtHp = simTime();
31     } else if (times[i] == lastServedWeightedDelta_rtLp) {
32         if (getQueueLength("rtLpQueue") > 0) {
33             send(new cMessage("schedulerMessage"), "rtLpQueueControl_out");
34             sent = true;
35         }
36         lastServed_rtLp = simTime();
37     } else if (times[i] == lastServedWeightedDelta_rtHp) {
38         if (getQueueLength("rtHpQueue") > 0) {
39             send(new cMessage("schedulerMessage"), "rtHpQueueControl_out");
40             sent = true;
41         }
42         lastServed_rtHp = simTime();
43     }
44 }
45
46 if (!sent) {
47     scheduleAt(
48         simTime()
49         + SimTime(
50             par("packetGenerationDelay").doubleValue(),
51             SIMTIME_US).trunc(SIMTIME_US),
52             readyToScheduleMessage);
53 }

```

Figure 5: The inner workings of the Weighted Round Robin scheduler

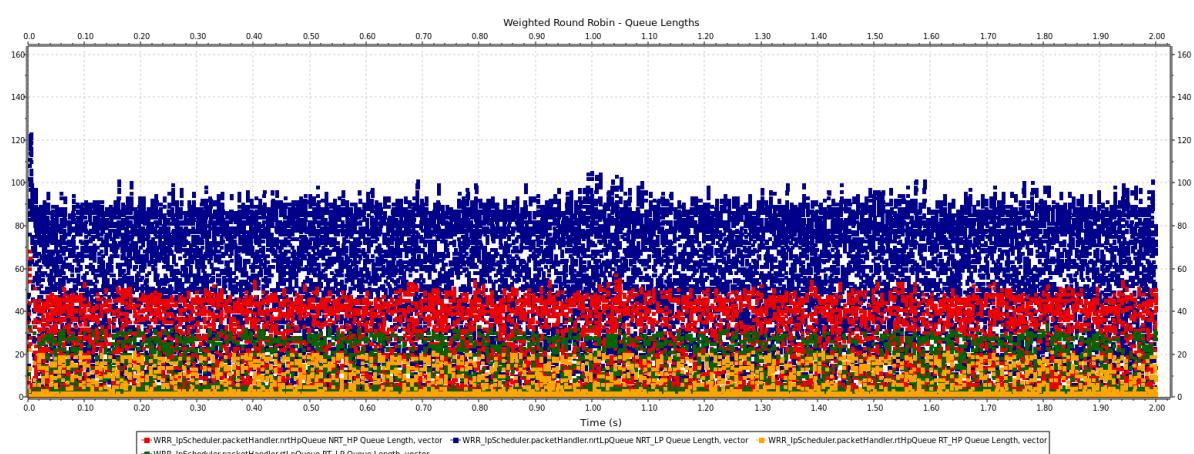
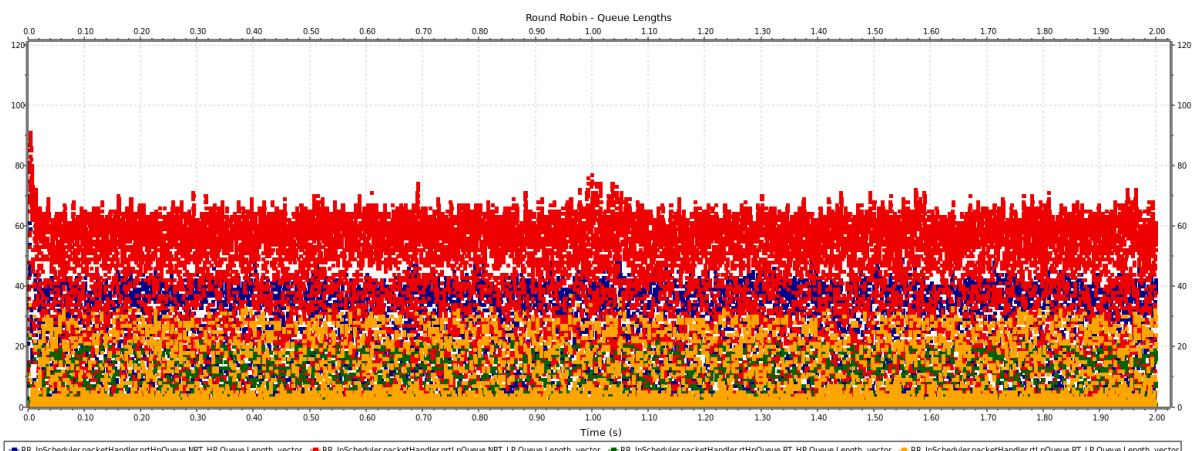
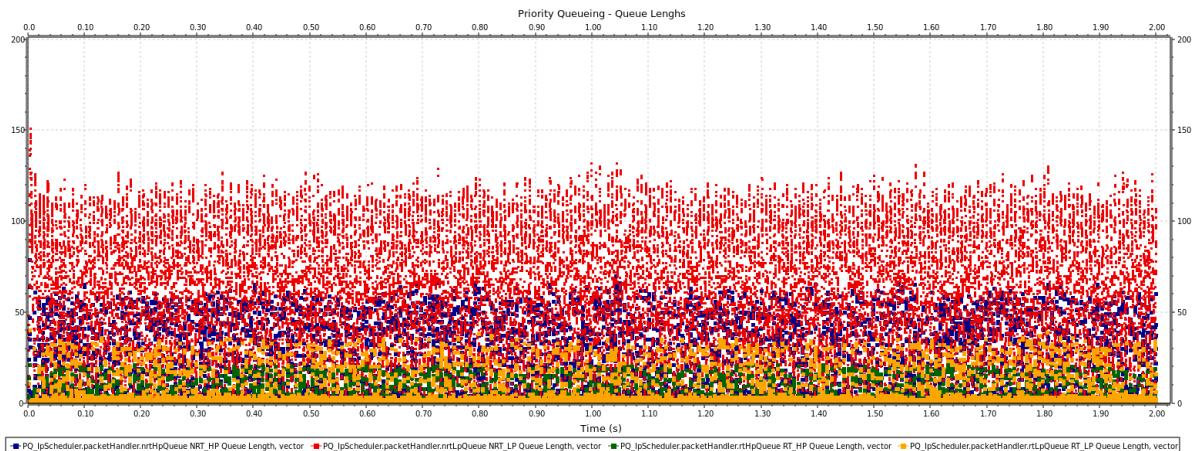
4 Comparative analysis

The following comparative analysis covers all three scheduling algorithms, namely Priority Queueing, Round Robin, and Weighted Round Robin. The system configuration at the time of the benchmarks is:

- **50** nrtLp Users, **40** nrtHp Users, **30** rtLp Users, and **20** rtHp Users
- for the WRR algorithm, the **weights** are **1**, **2**, **4**, and **8**, for nrtLp, nrtHp, rtLp, and rtHp respectively
- the **IP packet size** varies uniformly between **100** and **1500 bytes**

- the scheduler sends packets to the sink with a **datarate of 1Gbps**
- the **packet generation delay** varies triangularly between **900us** and **1100us**, with a peak at **1000us**, corresponding to an average upload rate per user of about **750KBps**

4.1 Queue Lengths



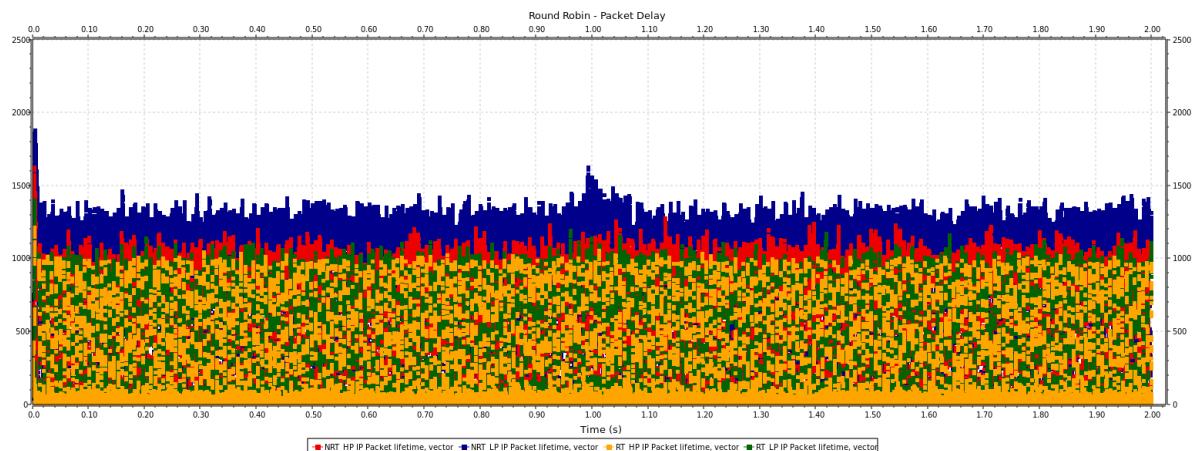
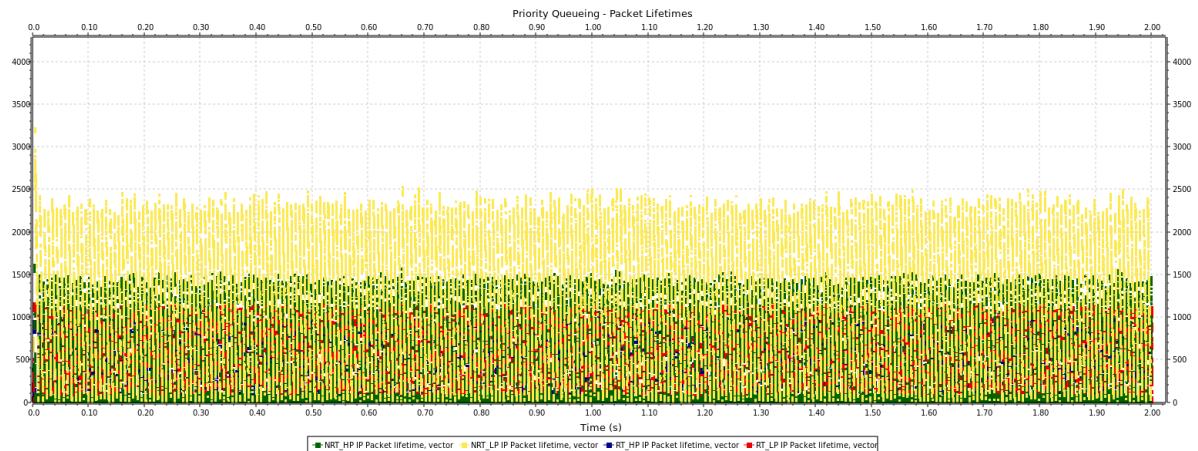
As we can see in the graphs above, the queue lengths form a nice layered structure, with each priority class having its own band of queue lengths. The lower priority users have higher queue lengths, and the higher priority ones, lower queue lengths.

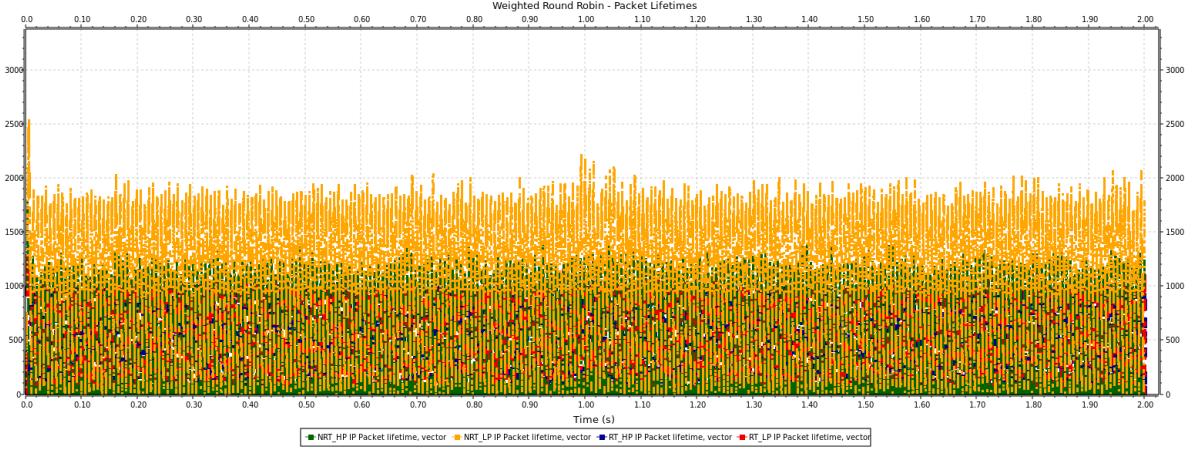
Below, you can find a table with the mean queue lengths for the four classes of users:

	nrtLp	nrtHp	rtLp	rtHp
Priority Queueing	59.56	10.77	4.46	2.54
Round Robin	44.27	19.69	8.00	4.03
Weighted Round Robin	54.12	14.44	5.42	2.88

The table shows that, when it comes to the queue lengths, the Priority Queueing scheduler is superior, while the Weighted Round Robin scheduler ranks second.

4.2 Packet Lifetime





The packet lifetime charts also seem to show a layered structure, just like the queue lengths, but this time the layered structure is a little more subtle, the differences being smaller. Also like before, the lower priority users show higher packet delays, while the high priority users show lower delays. It should be noted that the graphs show the packet delay in microseconds.

Below you can find a table with the mean packet lifetimes for the four classes of users:

	nrtLp	nrtHp	rtLp	rtHp
Priority Queueing	1194.64	254.85	129.31	86.49
Round Robin	884.45	476.96	264.28	161.53
Weighted Round Robin	1084.30	348.13	166.12	103.02

Again, as before, the Priority Queueing algorithm takes the lead, and the Weighted Round Robin is left in the second place. The Round Robin algorithm seems to favour lower priority users, but falls behind when it comes to the higher priority ones.

5 Conclusions

As the comparative analysis shows, in the presented case, the Priority Queueing algorithm seems to perform better than its counterparts. The Weighted Round Robin shows the second-best performance, and the Round Robin algorithm, although favouring lower priority users, falls behind in the other categories.

This, of course, does not match our expectations. We would have expected that the Weighted Round Robin algorithm to come in the first place, the Round Robin to come in second, and the Priority Queueing to be left behind by a long margin. This did not happen, I believe, for a number of reasons:

- the chosen system configuration (number of users, packet generation rate, etc.) matched a case in which the priority queueing performed better; if we would choose another configuration of parameters, the results would surely change
- the IP packet generation does not match a real-life scenario; the packets are generated at relatively constant rates, which quite rarely happens in real-life (normally, packets are generated in short bursts of packets, instead of evenly distributed packets)

- in the case of the Weighted Round Robin algorithm, the weights were chosen on a non-experimental basis; more experiments could reveal a more efficient configuration of weights than the one presented (or, perhaps, the algorithm could be made to make use of adaptive weights, that change according to factors such as network load)
- I do not exclude the possibility that the other algorithms could suffer improvements, which could tip the balance in one direction or another

6 References

- (1) http://staff.cs.upt.ro/~todinca/cad/IP_scheduling.html
- (2) <http://staff.cs.upt.ro/~todinca/TPAC/tpac.html>
- (3) <https://doc.omnetpp.org/omnetpp/manual/>
- (4) <https://docs.omnetpp.org/tutorials/tictoc/>
- (5) <https://doc.omnetpp.org/omnetpp/api/>