Radio system with location dependent errors

Nandor Csereoka, 4th Year – CTI-Eng – CAD, 2020-2021

Problem statement

Consider a radio system consisting of K users (e.g. K=3) which transfer data.

Between the moments t1 and t2, one of the users has a very bad radio link, with so many errors that the scheduler will not allocate radio resources (radio channels) to that user.

Compared to an ideal system (i.e., without errors), the user affected by (location dependent) errors will be lagging, which means that the amount of data transfered by that user is with LAG smaller than the data transfered by its correspondent in the ideal system.

After the moment t2 the user will have again a good radio link, and its weight in the scheduling algorithm has to be increased, with the purpose to reduce the value of LAG to zero, or very close to zero.

LAG is defined as the difference between the amount of data transdered by a user in the ideal system, and the amount of data transfered by the same user, in the real system.

Like in the previous problem, you can start from the fifo model from samples.

We can consider either a discrete scheduling algorithm (e.g. WRR - Weighted Round Robin), or you can consider that each user has a transfer rate equal with 1/K of the capacity C of the network (e.g. C= 1 Mega bits per second).

Try to adjust the weight of the user affected by errors (respectively its transfer rate) on the expense of the other users.

Minimal requirements

Implement the simulation model using a non-adaptive scheduling algorithm (e.g. Weighted Round Robin).

Extended requirements

Adapt the scheduling algorithm in order to allow the lagging user to reduce its lag. For example, you can increase its weight in the WRR algorithm.

Compare the performance of the scheduling algorithm without adaptation with the performance of the scheduling algorithm with.

(Source: http://staff.cs.upt.ro/~todinca/cad/Error_sched.html)

Getting started

Analysis of scheduling algorithms

We'll explore several scheduling algorithms in this analysis. But first, let's settle some terms.

Terms

- *User*: Module that generates messages periodically;
- *User count*: The number of users
- *Unlucky user*: user which loses their connectivity (i.e. won't be able to forward its messages) for a certain period of time;
- *Sink*: Module which receives all the messages generated by the users;
- *Scheduler*: Module which controls how many messages a user will be able to forward to the sink in a scheduling cycle, given a specific scheduling algorithm;
- *Scheduling algorithm*: a plan which decides how many channels should be allocated to a user;
- *Weights*: An array of values which indicates how important are a user's packets;
- *Channel count*: A value showing how many channels can be distributed among users.

Scheduling algorithms

- Dummy
- Weighted Round-Robin
- First come, first served

The dummy scheduler

Let's say that we have **N = 4** users, our array of weights looks like this: **w = [1, 2, 3, 4]** and we have **C = 30** channels to distribute.

sum(w[i]) = 1 + 2 + 3 + 4 = 10

This means that for a weight of 1, a user will receive factor = 1 x (30/10) = 3 channels.

So, user[0] will receive **c[0] = w[0] x factor = 3,** while the others will receive c[1] = 6, c[2] = 9, c[3] = 12.

The Dummy algorithm will allocate the above mentioned channels each scheduling cycle - even if **user[k]** did not generate **c[k]** messages. This can lead to channels which are not used - and users with a low weight might get several messages stuck in their queue.

Let's take a very similar example. Say we have N = 4 users, an array of weights **w = [1, 2, 2, 3]** and C = 30 channels to distribute.

sum(w[i]) = 1 + 2 + 2 + 3 = 8

This means that for a weight of 1, a user will receive **factor = 1 x (30/8) = 3.75** channels.

So, user[0] will receive **c[0] = w[0] x factor = 3.75**, while the others will receive c[1] = 7.5, c[2] = 7.5, c[3] = 11.25.

These, should be integer values, of course, so we take the integer parts and keep the remaining difference **c = [3, 7, 7, 11]**.

The remaining **30 - (3 + 7 + 7 + 11) = 2** channels are going to be distributed one by one, starting from the first user - we'll obtain **c = [4, 8, 7, 11]**.

These values change only when the unlucky user loses their connection. Their weight gets set to zero, triggering a change in the channels allocated. Let's go back to our first example.

C = 30; w = [1, 2, 3, 4]

If user[3] loses their connection, the weights become: **w = [1, 2, 3, 0]**, the allocated channels: **c = [5, 10, 15, 0]**.

Once the unlucky user comes back online with a changed weight e.g. **w[3] = 9**, the weights become: **w = [1, 2, 3, 9]**, then the allocated channels: **c = [2, 4, 6, 18]**.

The improved dummy scheduler (queue-aware)

The Queue Aware algorithm will also taken into account the length of each user's queue.

At first, we do a naive Dummy allocation. We'll probably find a user which produced less messages then they are able to send. We keep count of such users and the remaining channels that they leave behind.

Let's take the previous example: **C = 30, w = [1, 2, 3, 4], c = [3, 6, 9, 12]**.

Let's say that these are the queue lengths: **q = [9, 6, 10, 2]**.

We can see that user[0] and user[2] exceed their allocated channels, user[1] hits just enough and user3 produced much less.

Allocate all those channels that can and should be allocated, that is **c' = [3, 6, 9, 2]**.

We are left with 10 more channels waiting to be allocated (the last user only consumed 2 out of its allocated 12).

We'll do another dummy allocation, but keeping in mind those clients that have been fully served and reduce their weight to zero.

C' = 10, **w' = [1, 0, 3, 0]**, and then accordingly **c'' = [2.5, 0, 7.5, 0]** or rather **c''= [3, 0, 7, 0]**.

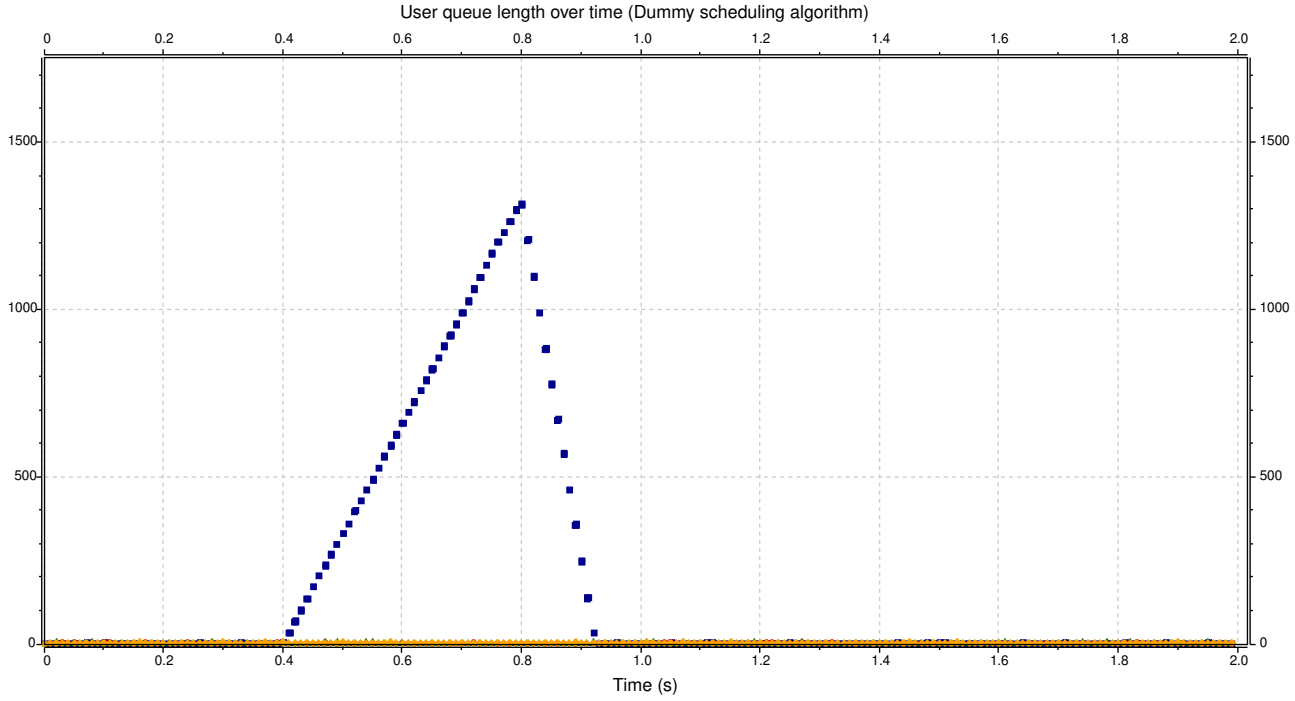
So now, for the final allocations - **c[k] = c'[k] + c''[k]**:

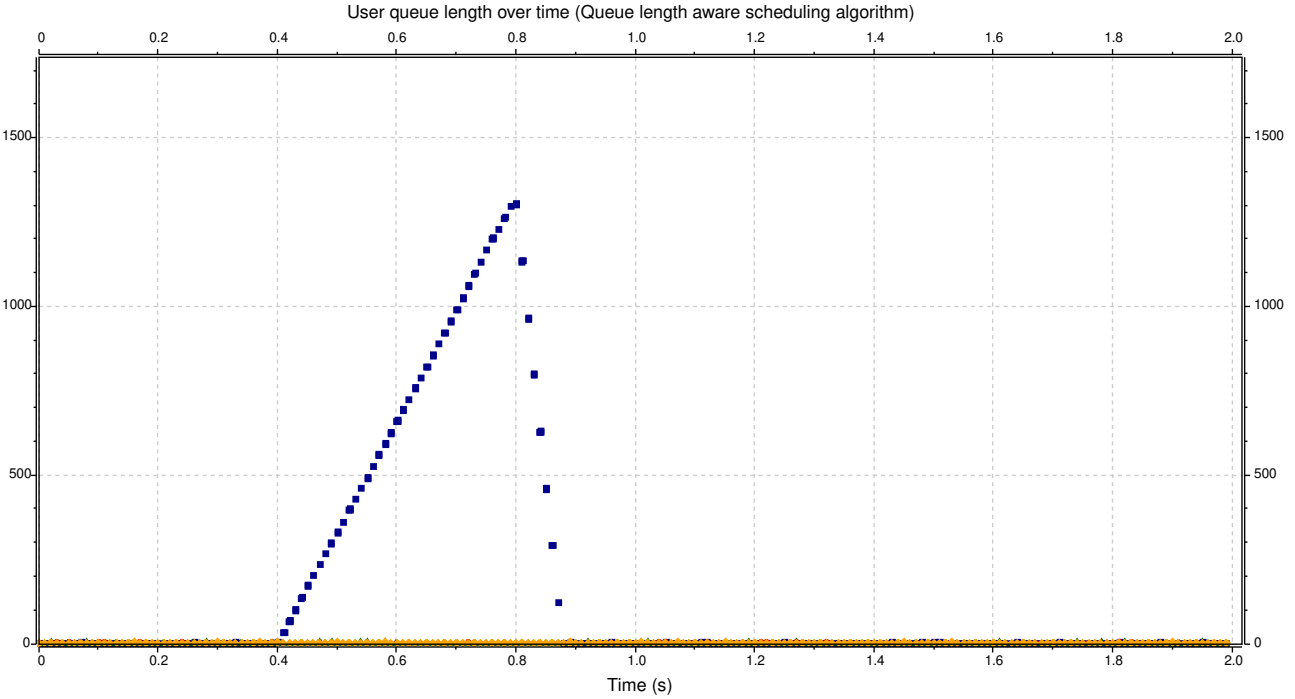
c = [6, 6, 16, 2].

We can continue this process until we get to an optimal resource allocation. This algorithm will do the optimization only once.

Observation! The problem with this algorithm (and of course the dummy one) is that it prioritizes the first users. In a scenario in which the channels which can be allocated is much lower than the amount of messages the users generate, the unfortunate users at the end of the user array will rarely get any allocated channels.

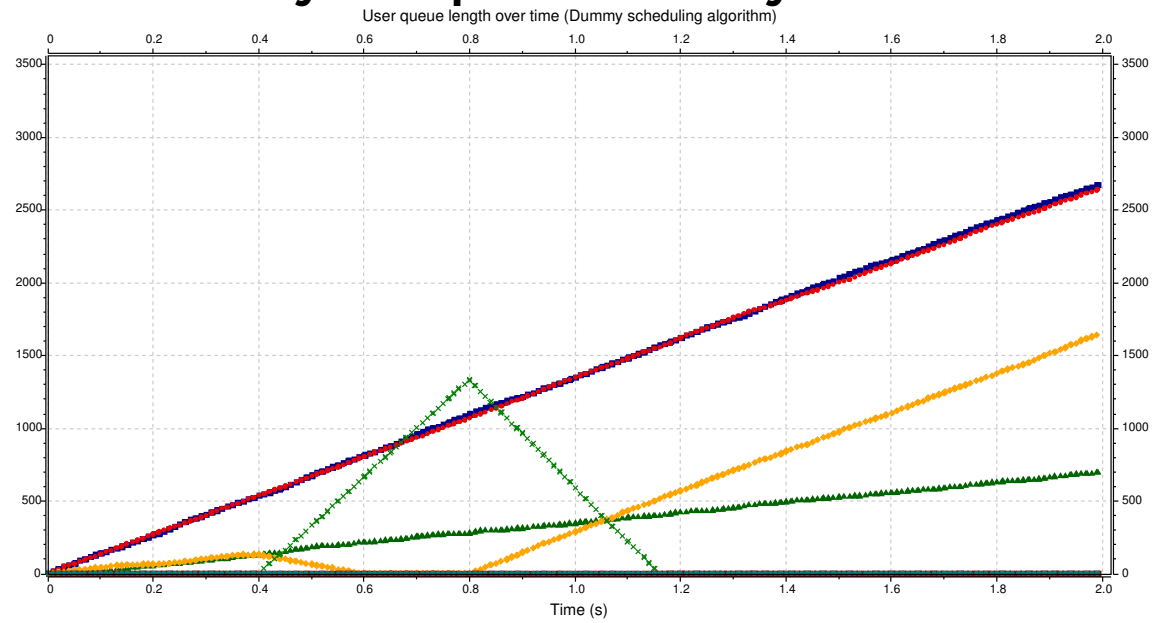
Dummy vs improved dummy – 4 users

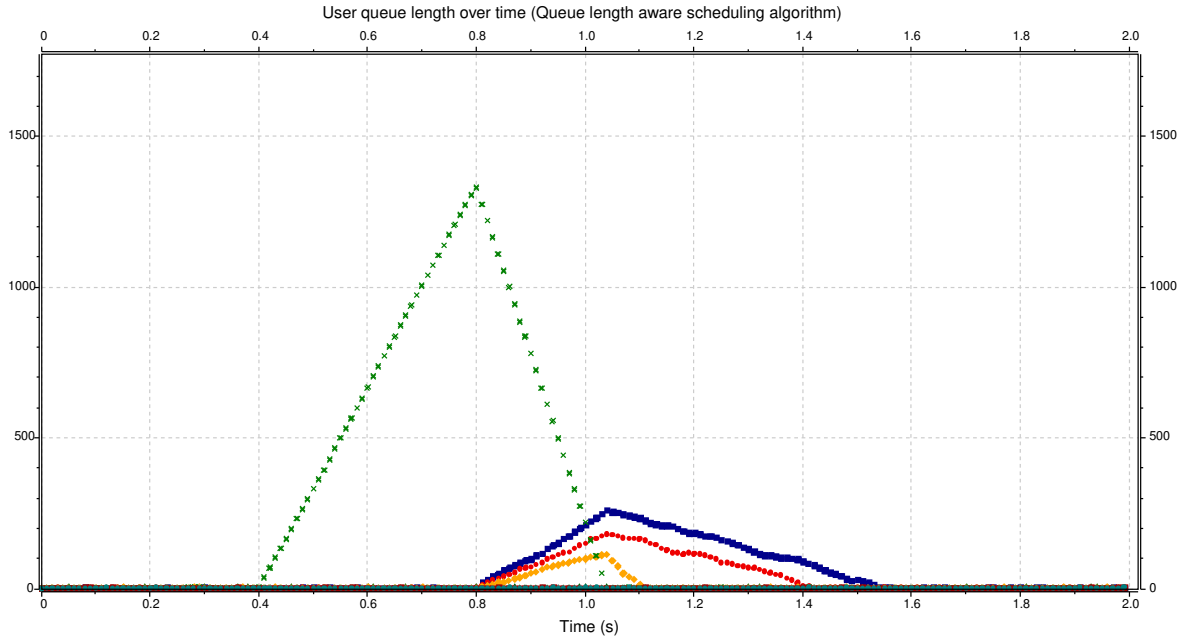




Due to the number of channels available and the packet generation interval, we don't see a clear advantage of the Queue Aware algorithm in this case.

Dummy vs improved dummy – 8 users





We can clearly see the problems that come with constant allocation and not taking into account user queue lengths when using the Dummy algorithm. The Queue Aware algorithm shows much better results.

We can see that when the unlucky user reconnects - **0.8s** - and is assigned a much greater weight, some of the other users get left behind. This continues until the unlucky user gets 'up-to-date' - after the 1s mark.

At the same time, we can also see the formation of a top for the other users. After this, we see a normalization in which the lengths of the queries return to the values from before the time the unlucky user's disconnect.

The Weighted Round-Robin Scheduler

The Weighted Round-Robin is a classic scheduling algorithm.

In our problem, this algorithm will look at the length of our users' queues and considering the number of channels which can be allocated, it will allocate each user a number of packets at a given time.

Round-Robin because we cycle through our users until the allocable channels are depleted.

Weighted because we give each user a particular priority.

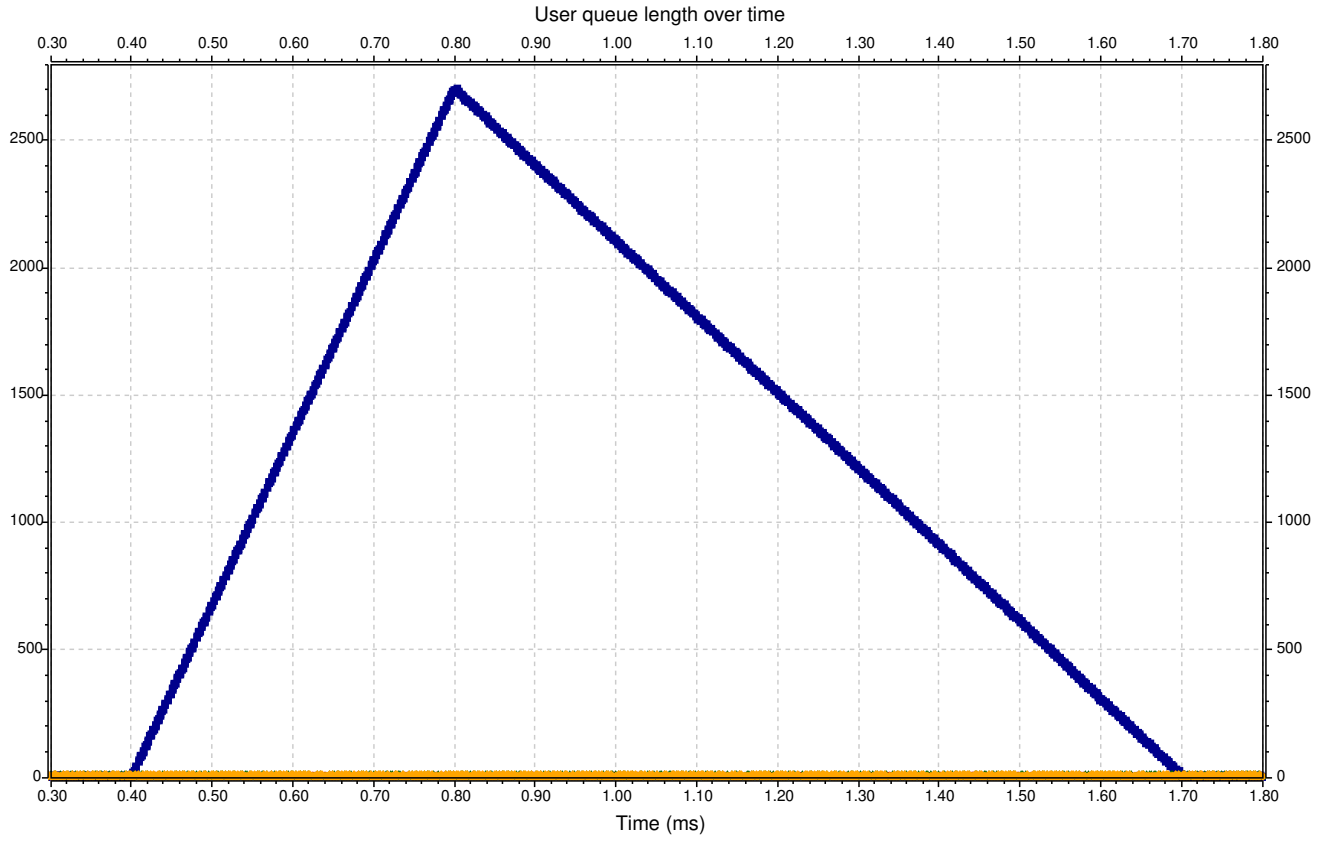
Weighted Round-Robin – 4 users

We follow the recommended conditions:

- a scheduling cycle of 1 ms (an LTE network);
- 30 channels which can be allocated.

For this analysis, we chose the following weights for the users: **w = [1, 2, 4, 8]**.
The user that gets disconnected is the user with the weight of 1.

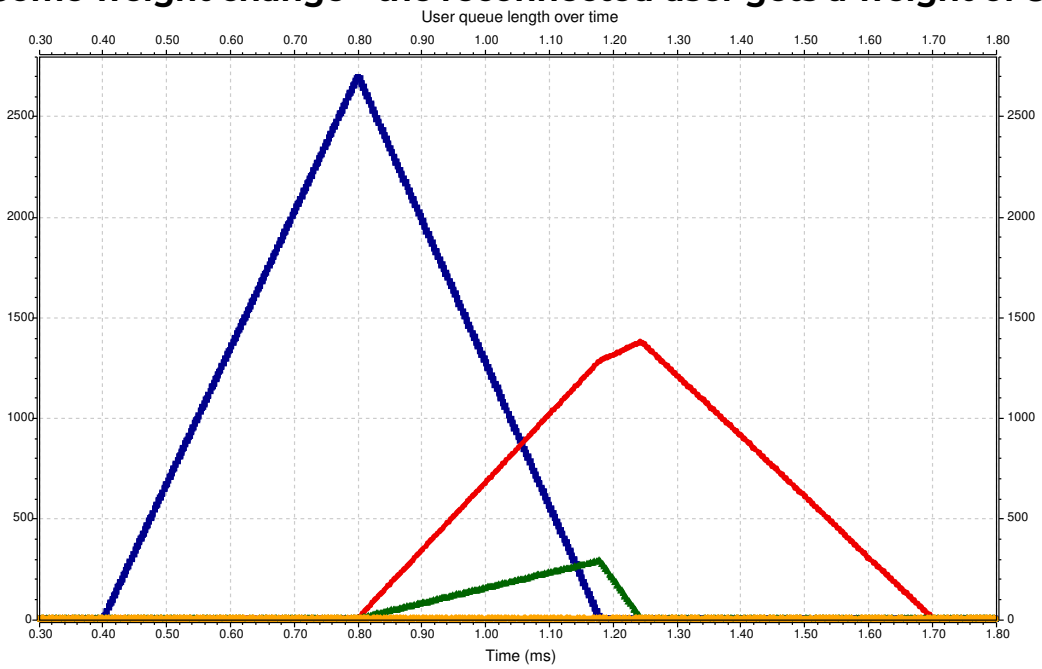
No tinkering - the weight of the user remains the same after reconnecting



The other uses don't get affected.

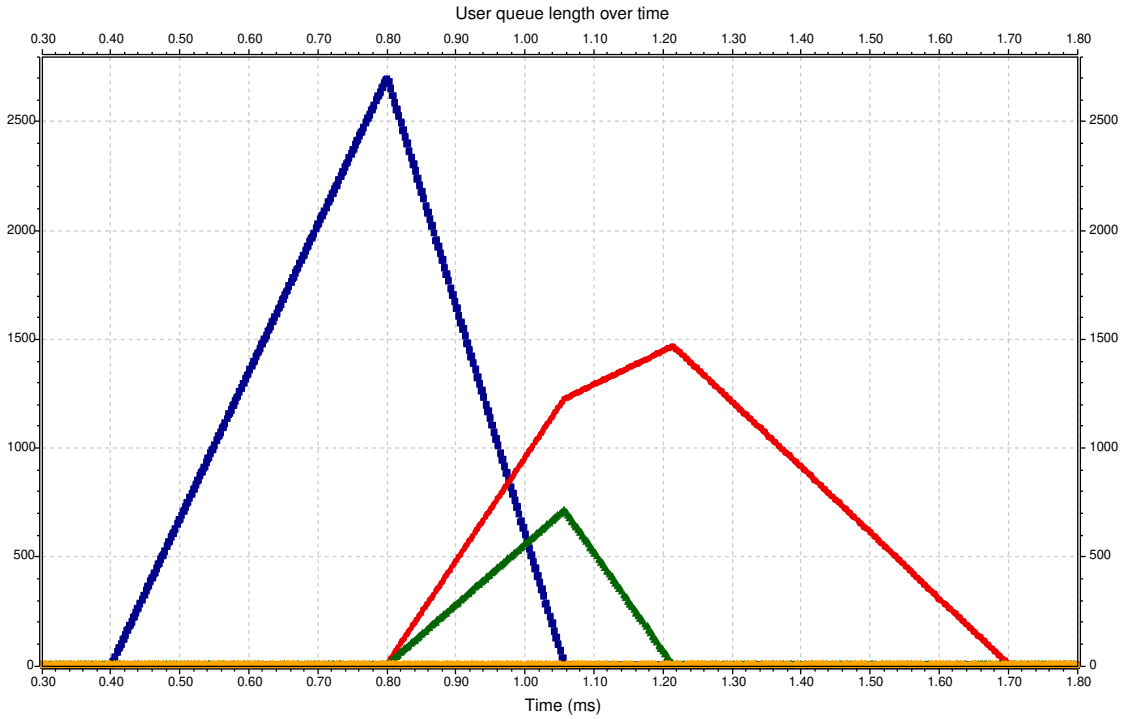
Weighted Round-Robin – 4 users

Some weight change - the reconnected user gets a weight of 8



The unlucky user gets the same weight as the user with the highest priority - a weight of 8. Users started to get affected.

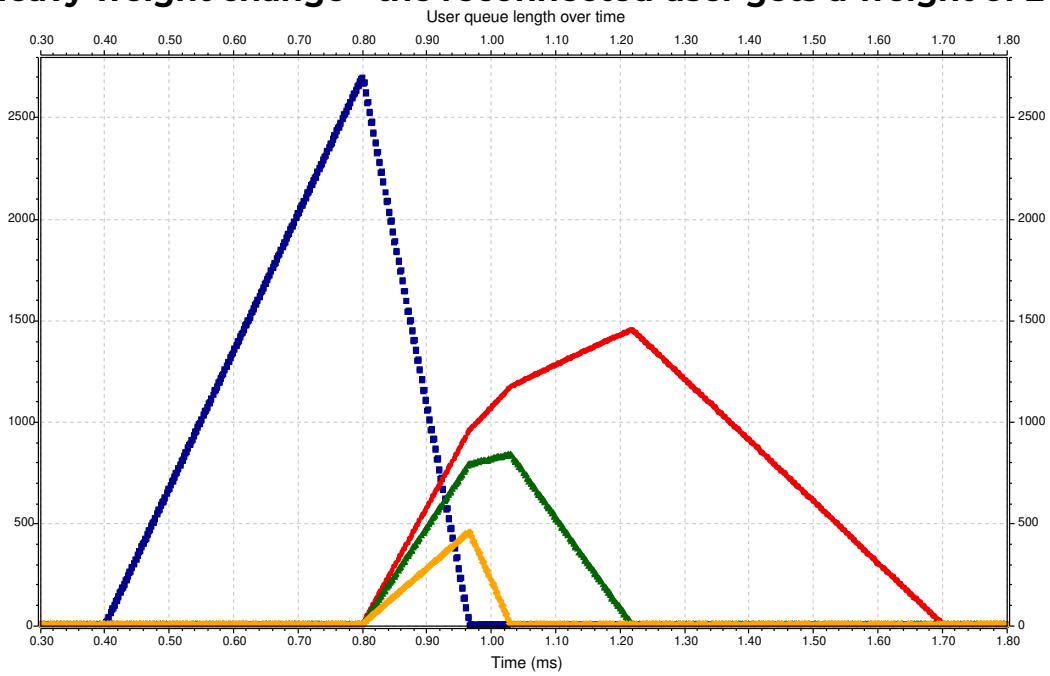
Medium weight change - the reconnected user gets a weight of 16



We see an increase in lag.

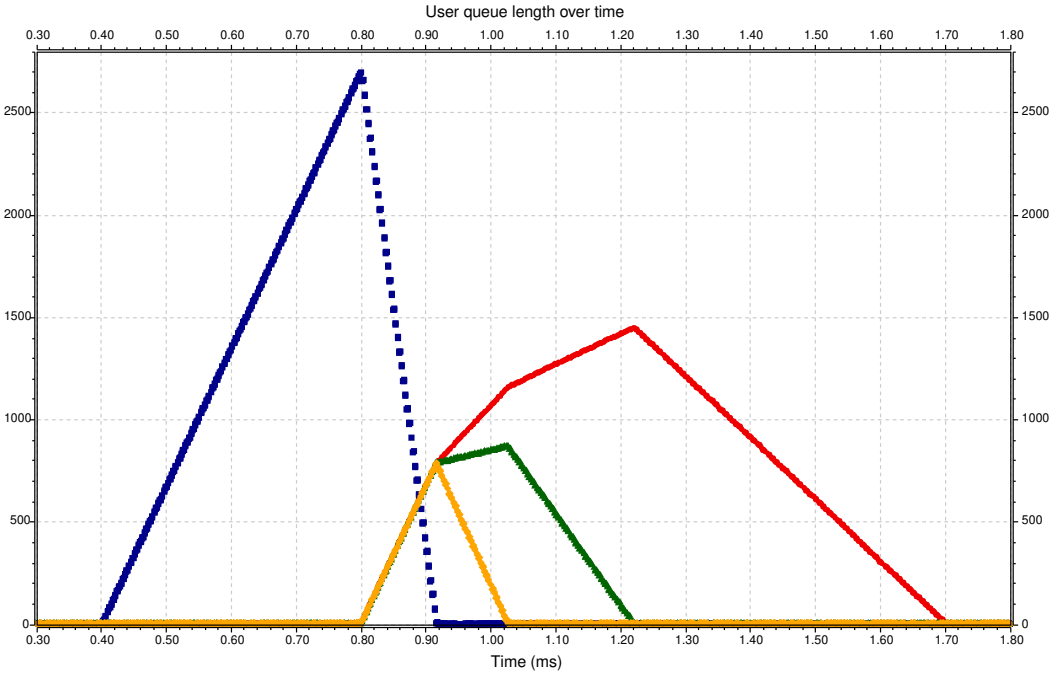
Weighted Round-Robin – 4 users

Heavy weight change - the reconnected user gets a weight of 24



At this point, all the other users gathered up lag.

Extreme weight change - the reconnected user gets a weight of 32



The newly allocated weight exceeds the number of channels which can be allocated.
We can see that the network won't allocate channels to any other user until the queue of the unlucky one gets depleted - causing heavy lag to everybody else.

Weighted Round-Robin – 8 users

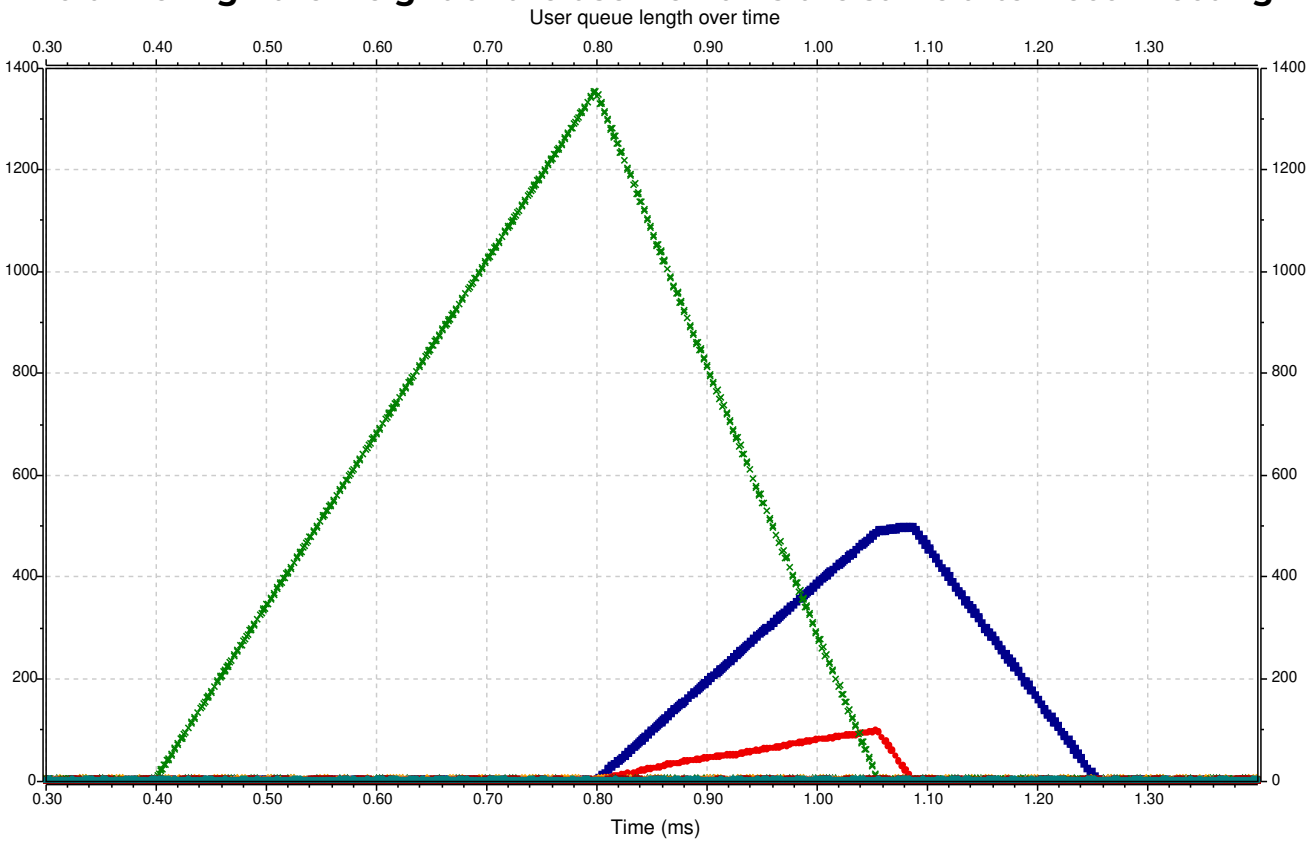
We follow the recommended conditions:

- a scheduling cycle of 1 ms (an LTE network);
- 30 channels which can be allocated.

For this analysis, we chose the following weights for the users:
w = [1, 2, 3, 4, 5, 6, 7, 8].

The user that gets disconnected is the user with the weight of 4.

No tinkering - the weight of the user remains the same after reconnecting

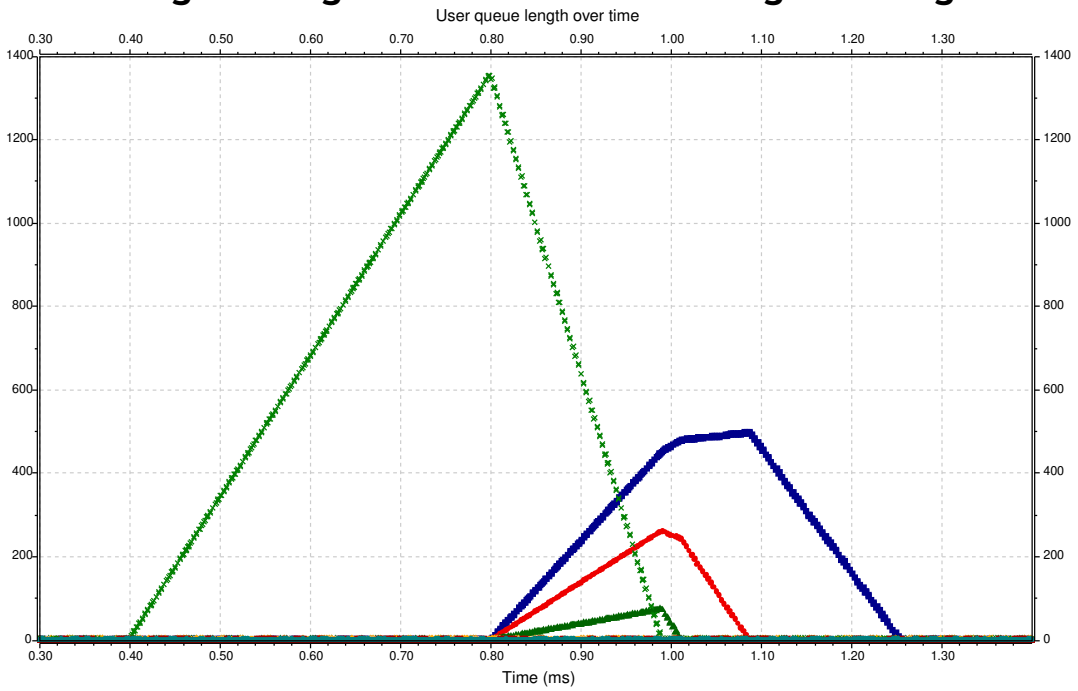


The users with the smallest weights get affected, which is expected.

In this case, only the ones with the weights 1 and 2.

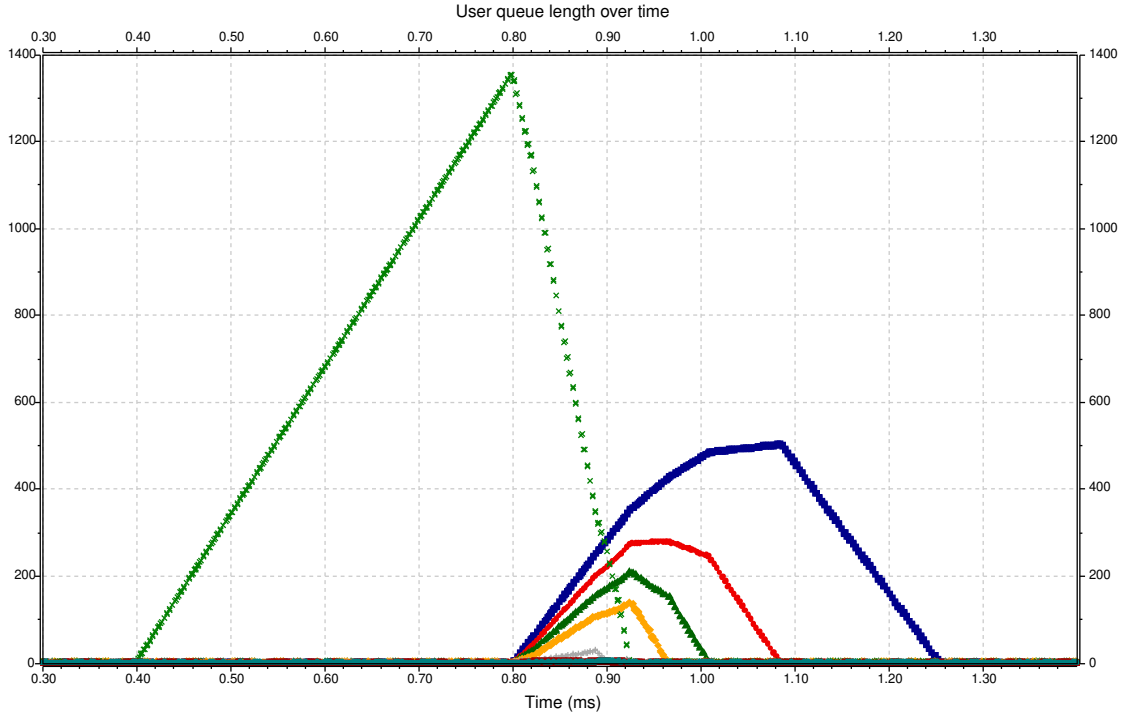
Weighted Round-Robin – 8 users

Some weight change - the reconnected user gets a weight of 8



The unlucky user gets the same weight as the user with the highest priority - a weight of 8. We can see that the user with the weight of 3 gets also affected by lag in this case.

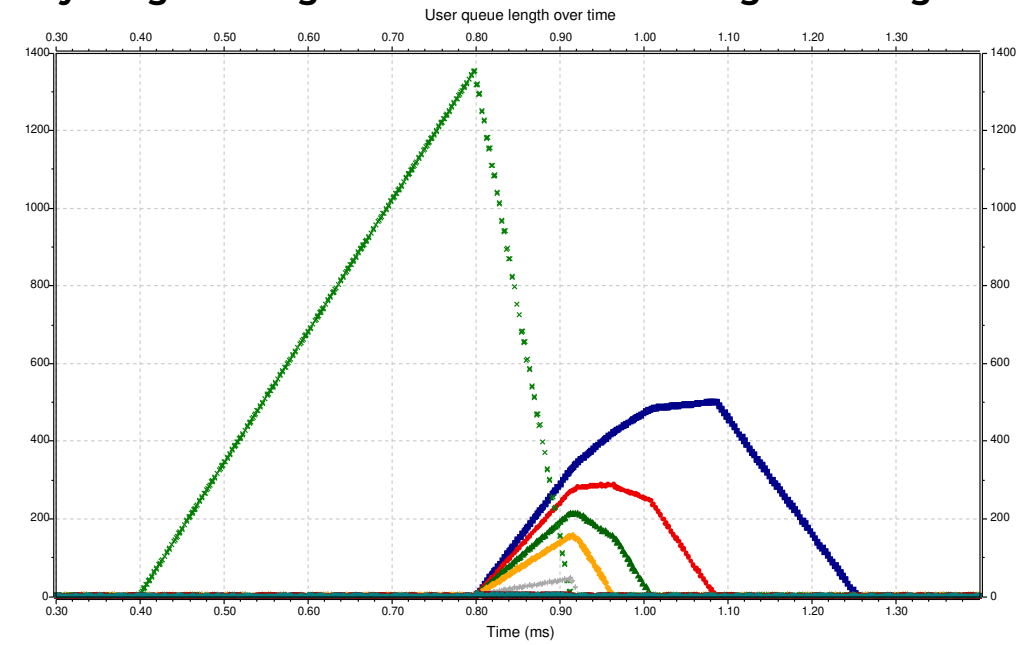
Medium weight change - the reconnected user gets a weight of 16



The other users start experiencing some lag.

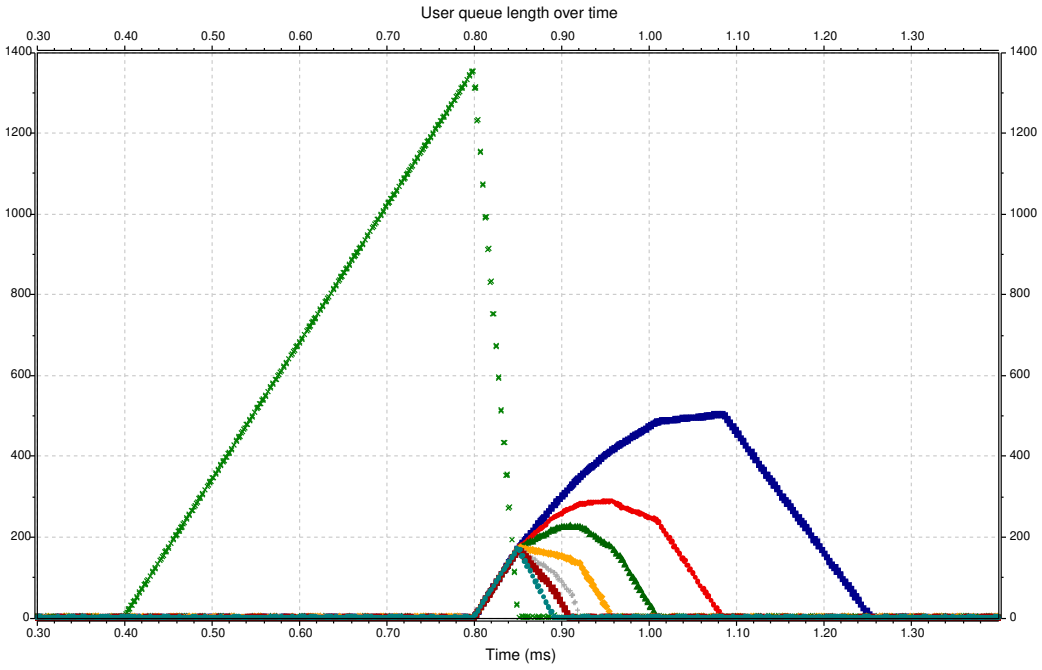
Weighted Round-Robin – 8 users

Heavy weight change - the reconnected user gets a weight of 24



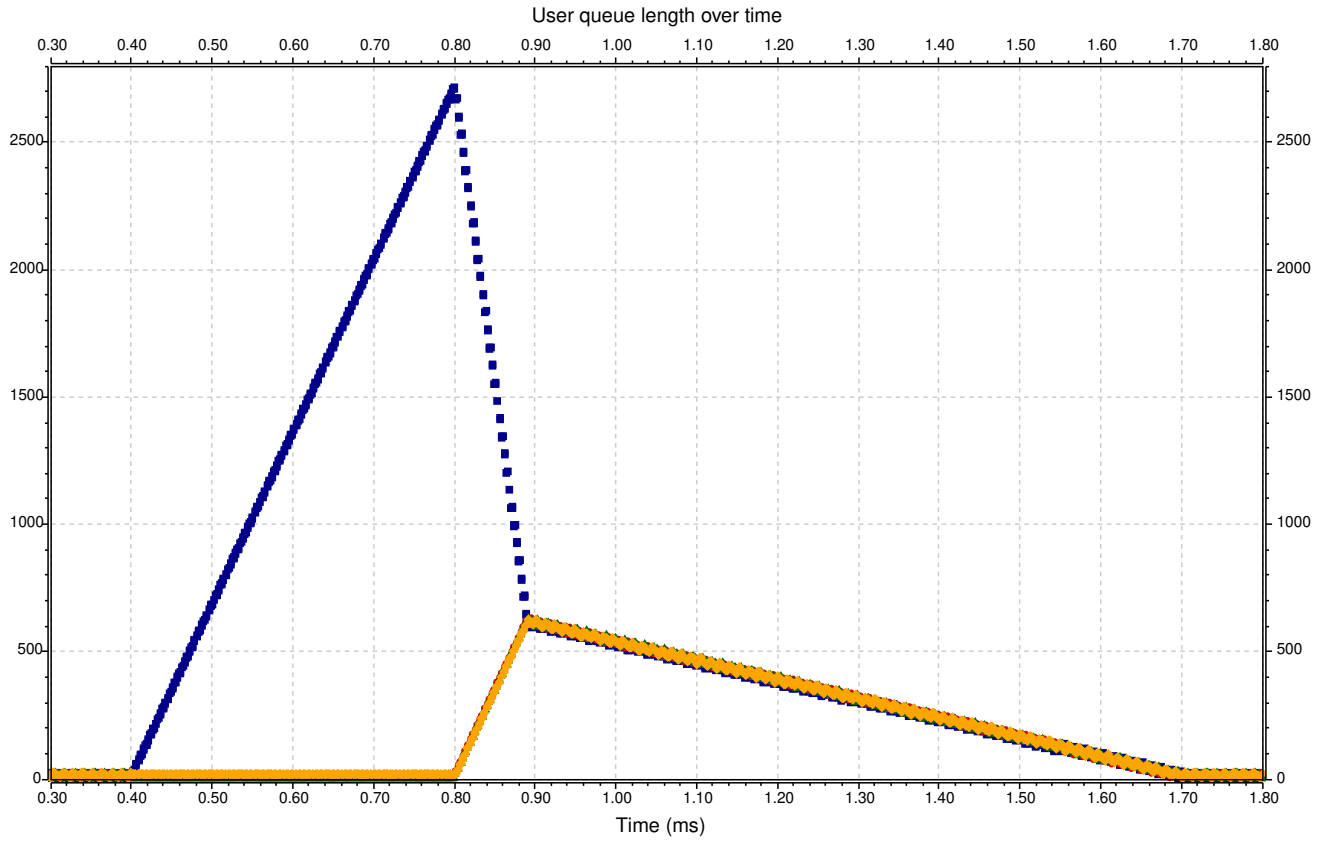
At this point, most of the other users gather up considerable lag.

Extreme weight change - the reconnected user gets a weight of 32



Yes, this clearly is unrealistic, especially because the newly allocated weight exceeds the number of channels which can be allocated. We can see that the network won't allocate channels to any other user until the queue of the unlucky one gets depleted - causing heavy lag to everybody else.

The *'First come, first served'* scheduler



The First in, first out (FIFO) or First come, first served (FCFS) takes into account only the time the packets entered a particular queue.

The algorithm will cycle through all the queues and look for the oldest message from the bunch. This goes on until the channels allocated are depleted.

In comparison to the Weighted Round-Robin, this algorithm not only affects all the users, but also keeps them 'behind' for a considerable amount of time - more than WRR does.

When the user reconnects, we can see the sudden prioritisation of their messages - due to them being stale sitting in the queue.

Their messages are going to be prioritised until we get to that point of balance where the two lines meet: the users' messages will be prioritised 'evenly', like at the beginning.