NORTEL NETWORKS
How the world shares ideas.

A Model for TI Egress Class Thresholds

Nicolae Santean
Network Traffic Engineering, 0S04

The Mathematical Model

The purpose of this paper is to present a simple method of setting up class thresholds for TI egress buffer. The problem is stated as following. Various traffic streams of different QoS classes share a common buffer space of size L. Packets are en-queued through dynamic memory allocation (from the common buffer space) in corresponding queues. Buffered packets are de-queued and transmitted via output channels by a scheduler following certain logic. In this scenario, packets belonging to intense traffic streams can eventually flood the common buffer, resulting in the degradation of other traffic flows. Therefore, an indexed set of class thresholds $\left\{t_j\right\}_{j\in\overline{0,n-1}}$ is used to control the buffer occupancy at the class level, where by n ($n \geq 2$) we denote the number of classes. An incoming packet is en-queued only if the buffer occupancy of its class does not exceed the corresponding class threshold, where by the class of a packet we understand the class of traffic stream to which the packet belongs. The buffer occupancy of a class is the total memory used by en-queued packets belonging to that class. The challenge we are taking is to find proper class thresholds such that competition among classes for buffer space does not degrade the targeted quality of service.

Traffic streams of the same class have associated similar QoS requirements. Therefore it makes sense to talk about QoS at the class level. The cumulated drop rate among all traffic streams belonging to a certain class defines the packet drop rate per class. Having given a maximal packet drop rate per each class, one can define the minimal/critical buffer space per class as following. Consider that in the systems exist only traffic streams of one class. What is the buffer size that ensures that cumulated packet drop rate does not exceed the maximal packet drop rate allowed for that class? This buffer size value defines the critical buffer space per class. Denote by $\left\{B_j\right\}_{j\in\overline{0,n-1}}$ the indexed set of critical buffer space per each class. In other words, a class j will always have available at least B_j buffer space, guaranteeing therefore that the drop rate does not exceed a certain maximal value.

The method presented in this paper derives class thresholds from the set of critical buffer space per class. Let consider a class j and assume that all the other classes fully use their buffer thresholds. It is clear that the class j must have left at least its critical buffer space B_j (for optimization purpose we consider that the left space is exactly the critical buffer space: in this way we maximize $\sum_{i=0}^{n-1} t_i$). Then the following system of equations holds:

$$\left\{ L - \sum_{\substack{i=0 \\ i\neq j}}^{n-1} t_i = B_j \right\}_{j\in\overline{0,n-1}}, \quad n \geq 2 \tag{0.1}$$

For $n \geq 2$ this system has a unique solution given by

$$\left\{ t_j = \frac{L - \sum_{\substack{i=0 \\ i\neq j}}^{n-1} B_i + (n-2) B_j}{n-1} \right\}_{j\in\overline{0,n-1}}, \quad n \geq 2 \tag{0.2}$$

The system (0.1) leads to an interesting observation: the sum of all except one class threshold cannot exceed the total buffer size:

$$\sum_{\substack{i=0 \\ i \neq j}}^{n-1} t_i \leq L \quad , \quad \forall\, j \in \overline{0, n-1} \tag{0.3}$$

Note that the sum of all class thresholds is allowed to exceed L and in the following we determine this excess. Consider the following substitution in order to simplify the class threshold formulas:

$$Ker_L = \frac{1}{n-1}\left(L - \sum_{i=0}^{n-1} B_i \right) \geq 0 \quad , if \left(Ker_L = 0 \right) then \left(t_j = B_j , \forall j \in \overline{0, n-1} \right) \tag{0.4}$$

The fact that Ker_L should be greater than or equal to zero makes sense since the total buffer space should be large enough to simultaneously accommodate the critical needs (critical buffer space per class) of all classes. The notation Ker_L is an abbreviation of "*Kernel*", reflecting the fact that is found in the expression of all class thresholds, as the following proves:

$$\left\{ t_j = Ker_L + B_j \right\}_{j \in \overline{0, n-1}} \quad , \quad \sum_{j=0}^{n-1} t_j = L + Ker_L \tag{0.5}$$

which is a compact representation of the class thresholds. Two new observations have been made along with the formulas. One is that if Ker_L is zero, than the class thresholds are equal to the corresponding critical buffer size per class (and their sum is exactly L), and the second is that the sum of all class thresholds exceeds the total buffer space exactly by Ker_L. Note also that the difference between any two class thresholds is equal to the difference between their corresponding critical buffer size per class:

$$t_j - t_i = B_j - B_i \quad , \quad \forall i, j \in \overline{0, n-1} \, , i \neq j \tag{0.6}$$

In addition, from (0.5) is clear that $t_j \geq B_j$, $\forall j \in \overline{0, n-1}$. The following inequality is derived from the definition of Ker_L:

$$Ker_L \leq \frac{L}{n-1} , or \quad Ker_L \leq \frac{100}{n-1} \% \ of \ L \tag{0.7}$$

which implies that the sum of all thresholds can not exceed $\frac{100}{n-1}\%$ of the total buffer space.

Summarizing, we have derived a system of equations that relates the critical buffer space per class with the class thresholds. We have derived the solution of this system and shown its properties. In the following we explore ways of determining the critical buffer space. Furthermore, we are interested in the interaction between the class threshold drop policy and the RED/tail drop.

The Critical Buffer Space

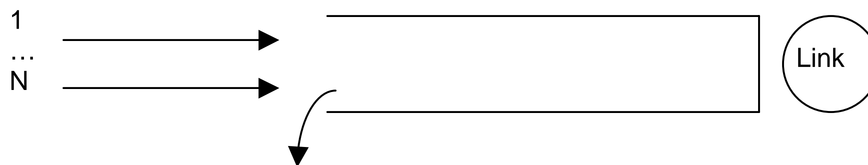
The critical buffer space per class can be found either experimentally or by analytical means – when enough information about the incoming traffic is provided. We present here four possible methods of finding this value.

The Recurrent Method

It is called recurrent because we use the method presented at the beginning of this paper in a recurrent fashion: we first use it globally, to find the class thresholds based on the critical buffer space per queue and then we use it within each class to find $tail(q)$ (the tail drop threshold per queue) or $\max_{RED}(q)$ (the RED parameter denoting the "full-drop" queue size). This method treats the class thresholds and queue drop levels in a unified manner. In addition, it reveals an important result: that for classes with a large number of queues the class thresholds are ineffective. Due to the importance of these results, the method is presented in detail in a separate section at the end of this paper.

Buffer Size Estimation for a Homogenous Traffic Mix

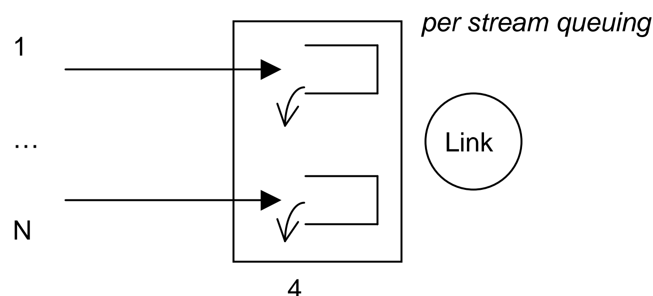
We isolate the traffic belonging to each class and study each class separately. Ignoring the queue per traffic stream, the problem has been simplified as following. We observe N traffic streams of similar nature (since they all belong to the same class) multiplexed via a common buffer into a link with a known service rate (the bandwidth allocation per class, eventually adjusted with a certain share of excess bandwidth).



We can assume an ON-OFF traffic model for each stream. Given the number of traffic streams and the buffer size we can estimate the probability of joining the buffer and the conditional cell-loss rate (see "Traffic Considerations in the Synthesis of an ATM-Based Network" by James Yan and Maged Beshay). Or in reverse, given these QoS values we can determine a minimal buffer size, which ensures that the quality of service is met. But this minimal buffer size is exactly the critical buffer space that we were looking for. For further reference on the topic see the Gibbens–Hunt EBR approach. Note that - in general - the average drop rate can be viewed as the average decrease of throughput.

The Queue Tail Drop – Rule of Thumb

We isolate the classes and treat each separately – as in the previous method. This time we take into consideration the queues per each traffic stream.



Let consider that there are N streams and that there is a tail drop policy at the queue level in order to limit the queue occupancy. Denote - as mentioned before - by $tail(q)$ the tail drop threshold per queue. Then the following rule of thumb (revealed courtesy of Stuart White) relates the queue tail drop threshold to the total buffer space:

$$tail(q) = \frac{available\ buffer\ space}{\sqrt{N}} \tag{0.8}$$

Note that if we consider the available buffer space in formula (0.8) to be an estimation for the required critical buffer space per class, then the following holds:

$$critical\ buffer\ space = \sqrt{N} \cdot tail(q) \tag{0.9}$$

This formula is particularly useful when we already have set queue tail-drop thresholds and we are required to further set the class thresholds. Note that some classes may have a RED drop policy implemented at the queue level. In that case, $\max_{RED}(q)$ can very well replace $tail(q)$ in the above formula.

The Experimental Method

The previous methods evaluate the critical buffer space by analytical means. One drawback of these methods is the tight set of assumptions they relay on. A way to decrease the number of assumption is to effectively simulate the behavior of the system in either common scenarios or worst-case scenarios and draw statistical conclusions. Even in this approach is advisable to first model simple systems comprising just one class and draw conclusions about the buffer occupancy in relation with some QoS parameters. Then, once we reach a good understanding of the critical buffer space per class, we can determine the class thresholds and verify the behavior of the integrated system by running a large-scale simulation.

RED/Tail Drop Interaction

The model for egress class thresholds presented in the first section does not capture any other packet drop policies. If RED/tail drop and/or memory balancing strategies are also used, extra caution is necessary to avoid unwilling interaction with each other. In the following we establish a "no-interference" condition between class threshold drop and RED/tail drop parameters.

We assume that - at a packet arrival – first the class threshold overflow and then the RED acceptance conditions are verified. It is clear that if a class threshold exceeds the sum of $\max_{RED}(q)$ for all queues, then the threshold will never be reached since the RED drop prevents each queue to grow over $\max_{RED}(q)$. Also, if the class threshold is smaller than the greatest $\min_{RED}(q)$ (RED parameter denoting the "full-acceptance" queue size) then RED will never drop packets for some queues, since the class threshold admission process will drop it first. For engineering reasons, we force the class threshold to be greater than the sum of all $\min_{RED}(q)$:

$$\sum_{q_i \in AD_x} \min_{RED}(q_i) < t_{AD_x} < \sum_{q_j \in AD_x} \max_{RED}(q_j), \quad for\ any\ AD_x\ class \tag{0.10}$$

In the case of tail drop per queue, $tail(q)$ replaces $\max_{RED}(q)$ obtaining:

$$0 < t_{EF} < \sum_{q_i \in EF} tail(q_i), \quad for\ any\ EF\ class \tag{0.11}$$

The Recurrent Method

As mentioned before, the importance of this method reside in the fact that it analytically determine the class thresholds and $tail(q)$, $\max_{RED}(q)$ and $\min_{RED}(q)$ per each queue in a unified manner. It also concludes that classes with a large number of queues do not effectively benefit from the class thresholds. The methodology is completely analytical, in the sense that it has no experimental/simulation components. In the following we describe the algorithmic steps of the method and we present each step in details.

The Recurrent Method - Algorithm

Step 1. Focusing on one queue, determine the critical buffer space per queue based on classical queuing models (for example using an M/M/1 system where, knowing the admissible delays and drop rate, we can determine the minimal buffer space which meets the QoS).

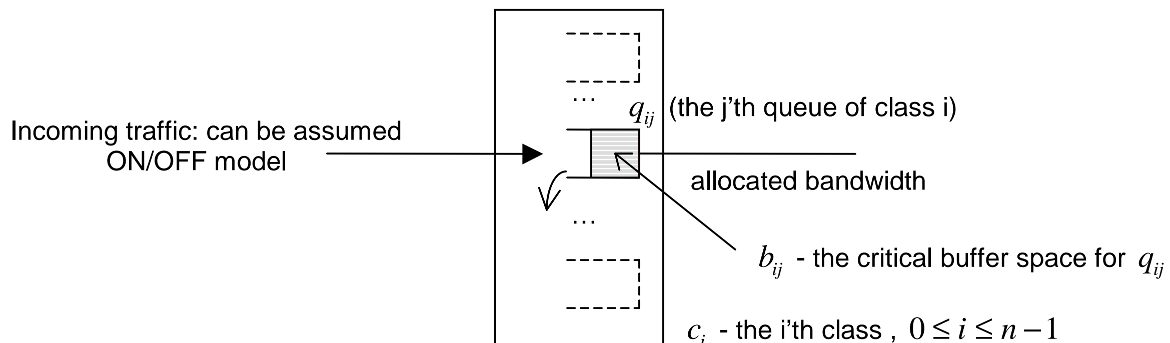
Step 2. Determine the critical buffer space per each class by adding up the critical buffer space of the corresponding queues. Note that it makes perfect sense to - later on - set the $\min_{RED}(q)$ to be exactly the critical buffer space per queue.

Step 3. Knowing the total buffer space and the critical buffer space per each class determine the class thresholds by solving the system of equations (0.1).

Step 4. Within each class: considering that the class threshold gives the available buffer and knowing the critical buffer space per queue we can use a system of equations similar to system (0.1) to determine $tail(q)$ or $\max_{RED}(q)$ per queue.

Steps of The Algorithm

Step 1. In this step we target the critical buffer space per queue. From the definition, the critical buffer space represents the minimal queue length that guarantees some predefined QoS values. The following information is assumed available: the statistical nature of the incoming traffic, the bandwidth allocated to the queue, the maximal admissible drop rate and the maximal admissible delay:



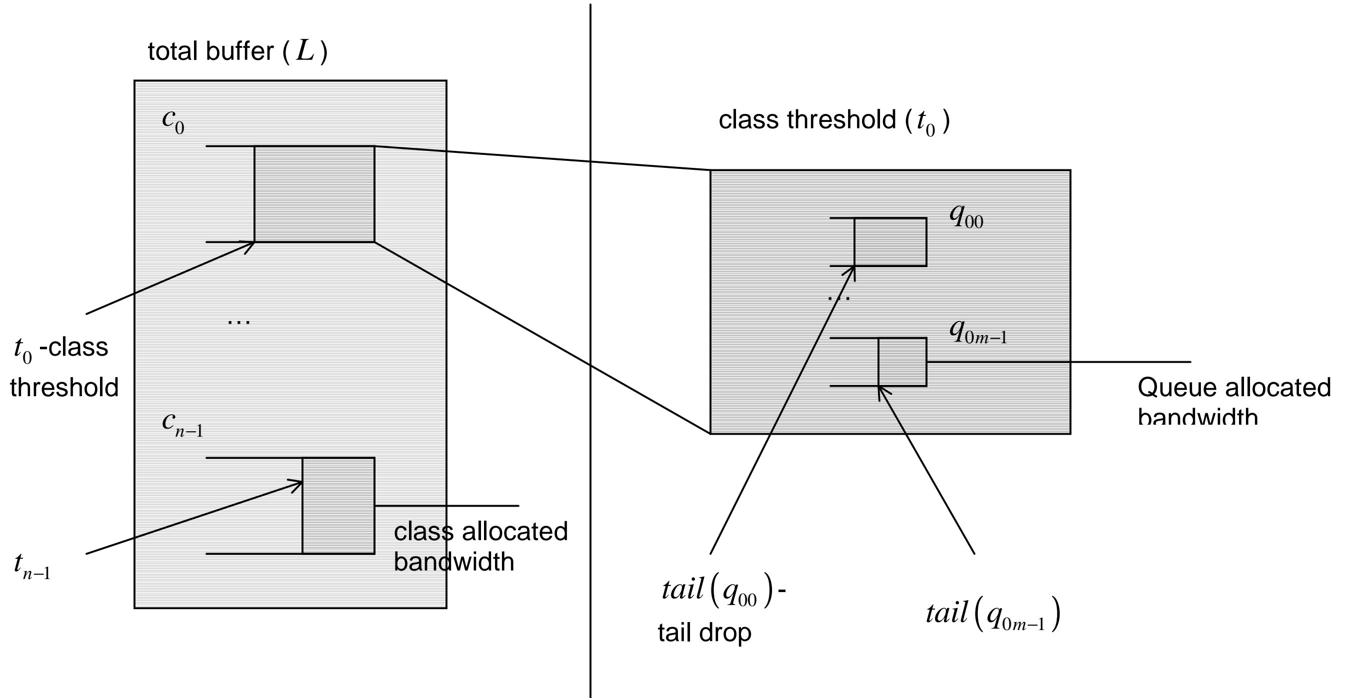
We have sufficient information to find b_{ij} – critical buffer space per queue – that meets the QoS. Observe also that b_{ij} will be a good candidate for $\min_{RED}(q)$ parameter for RED in the case of AD traffic.

Step 2. From the definition of the critical buffer space per queue, b_{ij} has to be guaranteed "no matter what". It is clear then that $\sum_{q_{ij}\in c_i} b_{ij}$ has to be guaranteed at the class level, in other words the class critical buffer space is the sum of all critical buffer spaces of all its queues. Therefore:

$$B_i = \sum_{q_{ij}\in c_i} b_{ij}\ ,\quad \forall i \in \overline{0, n-1} \tag{0.12}$$

Step 3. Moving our view to the class level, we observe that there is enough information for solving the system (0.1) and find $\{t_j\}_{j\in\overline{0,n-1}}$ - the set of class thresholds. Essentially, we found the critical buffer space for a class by studying its queues and then we solved system (0.1). But the problem is not finished yet. We will further use the critical buffer space per queue found in [Step 1] as a link between the class thresholds and the queue drop levels.

Step 4. Note that there is a perfect analogy between the buffer occupancy per class and the buffer occupancy per queue within a class. This analogy leads to the idea to recurrently use the method of [Step 3] to find the unconditional drop levels per queue (tail drop for EF and $\max_{RED}(q)$ for ADs). The following figure reflects this analogy:



<u>The analogy:</u>

total buffer limit	~	*class threshold limit*
class threshold	~	*queue tail drop or* $\max_{RED}(q)$
class critical space	~	*queue critical space*

On the left side of the above figure everything is known: total buffer limit, class critical space (found in [Step 2]), class thresholds (found in [Step 3]). On the other hand, on the right side we know the class threshold limit and the queue critical space (found in [Step 1]). It is clear now that applying a system of equations similar to system (0.1) we can find the queue tail drop. The details follow:

$$\left\{ t_i - \sum_{\substack{q_{ik} \in c_i \\ k \neq j}} tail\left(q_{ik}\right) = b_{ij} \right\}_{q_{ij} \in c_i} \quad , \quad i \in \overline{0, n-1} \tag{0.13}$$

For a class c_i with - say - m queues, $m \geq 2$, this system has a unique solution given by

$$\left\{ tail\left(q_{ij}\right) = \frac{t_i - \sum_{\substack{k=0 \\ k \neq j}}^{m-1} b_{ik} + (m-2) b_{ij}}{m-1} \right\}_{j \in \overline{0, m-1}} \quad , \quad m \geq 2 \tag{0.14}$$

Using a previous observation(0.7) for the solution of system (0.1) and recalling our analogy, is easy to show that the sum of tail drop levels for all the queues - $\sum_{q_{ij} \in c_i} tail\left(q_{ij}\right)$ - can not exceed t_i (the corresponding class threshold) by more than $\frac{100}{m-1}\%$. But then, when the number of queues m of class c_i is large enough, the sum of tail drop levels is almost equal to the class threshold:

$$\sum_{q_{ij} \in c_i} tail\left(q_{ij}\right) \underset{m \to \infty}{\longrightarrow} t_i \tag{0.15}$$

This makes the use of class thresholds ineffective for classes with a large number of queues.

Concluding, along with the class threshold per each class we have found the tail drop for EF queues and/or $\max_{RED}\left(q\right)$ and $\min_{RED}\left(q\right)$ for AD queues. Note that $\max_{RED}\left(q\right)$ and $\min_{RED}\left(q\right)$ are the only RED parameters determined by this method and that the remaining RED parameters should be determined by other means in strict relation to the TCP feedback mechanism.

A note concerning system (0.1)

As mentioned in the first section, system (0.1) should be actually viewed as a system of inequalities:

$$\left\{ L - \sum_{\substack{i=0 \\ i \neq j}}^{n-1} t_i \geq B_j \right\}_{j \in \overline{0, n-1}} \quad , \quad n \geq 2 \tag{0.16}$$

If we choose equations instead – as we did along this paper – we obtain a solution which maximizes the term $\sum_{i=0}^{n-1} t_i$. In fact this was our main goal since the purpose of class thresholds is to avoid the partitioning of the total buffer. The maximization ensures that we are as far as possible from a partition. The drawback of this solution is that it uniformly treats the classes when it gives them memory in addition to their critical buffer space. Most of the time this is an advantage since the method does not require extra class information and optimization criteria. Note that there is no stop of using the system of inequalities and linear programming techniques if we are in the possession of extra optimization criteria or constraints.

The following figure is a picture of the solution space of system (0.16) from the linear programming point of view. Three classes with identical critical buffer space values have been considered. The vector $\overrightarrow{OA}$ is the unique solution obtained by solving the system of equations (0.1) and the hexahedral solid $[OABCD]$ is the space solution of the system of inequalities (0.16) considering that we accept only positive solutions.