Exploiting Network Calculus for Delay-based Admission Control in a Sink-Tree Network

Luciano Lenzini, Linda Martorini, Enzo Mingozzi, Giovanni Stea
Dipartimento di Ingegneria dell’Informazione, University of Pisa
Via Diotisalvi 2, 56122 Pisa, Italy
{l.lenzini, l.martorini, e.mingozzi, g.stea}@iet.unipi.it

Abstract

In this paper we propose an admission control algorithm which is suitable for real time traffic in a sink-tree network in which per-aggregate resource management is in place. Every flow specifies its leaky-bucket parameters and a required delay bound, and the algorithm admits a new flow if a guarantee can be given that the delay bound for every established flow is not exceeded. The algorithm is based on a formula, derived through Network Calculus, that relates the worst-case delay of a single flow to the transmission rate provisioned for the aggregate and to the leaky-bucket parameters of flows at the ingress of the network. We describe the algorithm and the employed data structures, and we evaluate its complexity.

1. Introduction

The future Internet is foreseen to support the provision of reliable real-time services on a wide scale. A per-aggregate resource management is nowadays regarded as the solution for achieving such scalable service differentiation. A noticeable example is the Differentiated Services (DiffServ) [1] architecture, recently standardized by the IETF, in which flows traversing a domain are aggregated in a small number of classes or Behavior Aggregates (BA), and QoS is provisioned on a per-aggregate basis at each node. On the other hand, Multi-Protocol Label Switching (MPLS, [2]), that allows one to classify flows into forwarding equivalence classes (FECs) and to perform forwarding and routing on a per-FEC basis, is currently employed for supporting advanced traffic engineering schemes. Some recent work has focused on employing a sink-tree resource management scheme in both DiffServ ([3], [4]) and MPLS ([5], [6], [7], [8]) networks. Sink-tree resource management consists in partitioning a network in a set of logical trees, rooted at egress nodes and having ingress nodes as leaves, so that traffic directed towards an egress node is routed through the related tree. Furthermore, resources (i.e., bandwidth and buffer space) are provisioned on a per-sink-tree basis at each node. For instance, in MPLS, this is done by establishing multi-point-to-point Label Switched Paths, which also allows for saving label space compared to a point-to-point strategy. How to establish an optimal set of sink trees in networks of arbitrary topology has also been the focus of recent researches [4], [7], [8]. Hereafter, we call sink-tree network (also called accumulation or multipoint-to-point) one in which a sink-tree resource management scheme is in place. In a recent work [9], [10], a methodology was devised for computing the worst-case delay for single flows in a sink-tree network. The methodology is based on Network Calculus [11], [12], [13], [14], a theory for deterministic network performance analysis, and it allows one to relate the worst-case delay of single flows to the amount of resources provisioned at each node for its sink tree and to the traffic at the ingress. As such, it can be used as a starting point for devising traffic management schemes suitable for real-time traffic in sink-tree networks. In this paper, we devise an admission control for real-time traffic in sink-tree networks. More specifically, the algorithm checks whether leaky-bucket shaped flows requesting admission can be guaranteed a required end-to-end delay bound and no losses (save those due to physical errors) given the amount of available resources. The admission control decision is based on the worst-case delay formula derived in [10]. Thus, it is optimum, meaning that it only rejects a candidate flow if a chance actually exists that doing so would make a bit of its (or some established flow’s) traffic violate its required delay bound. However, an admission control test for a flow might entail a large amount of computations. In fact, we show that the worst-case delay of a flow also depends on the leaky-bucket parameters of other flows in the sink tree. Therefore, the admission of a new flow requires recomputing the worst-case delay for all the established flows, so as to check if their delay bound requirements are still satisfied. Therefore, we focus on evaluating the complexity of the admission control algorithm. More specifically, we describe the data structures and procedures...
required for testing the admission of a flow, from which we derive the overall complexity of the admission control scheme. We show that the complexity is \(O(N \cdot D + \log d)\), where \(N\) is the number of nodes in the network, \(D\) is the maximum path length and \(d\) is the number of flows at one ingress node, which makes the algorithm scalable.

The rest of the paper is organized as follows: Section 2 introduces the system model and states the problem. In Section 3 we describe how to compute the delay bound for a single flow in a sink tree, while in Section 4 we describe the admission control algorithm. Section 5 describes the related work, and conclusions are drawn in Section 6.

2. System model and problem statement

We focus on an arbitrary topology network domain (e.g., a DiffServ or MPLS domain), modeled as a network of switching elements (i.e., IP routers), connected by links. Following standard practice, we define as a node a network element which adds variable delay to traffic, e.g., the buffering and scheduling logic managing an output link of a router. Nodes are tagged with a unique label, which is an integer number. In the network, a sink-tree resource management scheme is employed, meaning that:

- a set of logical sink trees, each one rooted at egress nodes and having ingress nodes as leaves, is created.
- Traffic directed to an egress node is marked as belonging to the sink tree rooted at that node \(^1\) at each ingress (e.g., by assigning an ad-hoc DiffServ Code-Point or MPLS label).
- At each node, resources are provisioned on a per sink-tree-basis. More specifically, one FIFO queue is reserved for traffic belonging to each sink tree. As far as scheduling is concerned, the only constraint we put is that the service it guarantees to a queue can be modeled through a rate-latency service curve [7]. Such a definition includes a large number of well-known schedulers, e.g., Packet Generalized Processor Sharing [15], and Deficit Round Robin [16], which guarantee a minimum departure rate.

Thus, different sink trees – although sharing physical resources, e.g., the bandwidth of physical links – are independent of each other. Therefore, hereafter we consider only one such sink tree, which we model as a network of variable-rate FIFO servers. We denote with \(R_i\), \(\theta_i\) and \(S_i\) the guaranteed rate, latency and buffer space at node \(x\) for that sink tree.

A path \(P_i\) is a loop-free sequence of \(N_i\) nodes, from an ingress node to an egress node. In order to denote a node’s position in a path, we define function \(f_i(h)\) that returns the label of the \(h\)-th node in path \(P_i\), \(1 \leq h \leq N_i\), and function \(g_i(z)\) that returns the number of sequence of node \(z\) along path \(P_i\), \(g_i(\cdot) = f_i^{-1}(\cdot)\). Given two paths \(P_i\) and \(P_j\), \(i \neq j\), their traffic is aggregated at the first common node \(M_{ij}\). We then say that the two paths merge at that node, i.e. \(M_{i,j} = f_i(a) = f_j(b)\), for some \(a, b\) such that \(1 \leq a \leq N_i\) and \(1 \leq b \leq N_j\) and \(f_i(a-1) \neq f_j(b-1)\). Thus, we can use the ingress node label as a path subscript, i.e. \(f_i(1) = i\), without any ambiguity. Figure 1 shows a sink tree with six paths defined. For instance, it is \(M_{2,3} = f_2(2) = f_3(2) = 4\), and \(M_{3,8} = f_i(3) = f_3(2) = 10\).

![](image)

Figure 1. Paths in a sink tree

A sink tree is traversed by flows, i.e. distinguishable streams of traffic. A flow has a delay constraint, specified as a required end-to-end delay bound \(\delta\). Its arrivals are constrained by a leaky-bucket shaper, with a burst \(\sigma\) and a sustainable rate \(\rho\), at its ingress node. We assume that traffic is fluid, leaving packetization issues for further study. We want to devise an admission control test suitable for real-time traffic in such a network. More specifically, the test has to admit a new flow if and only if:

i) that flow can be guaranteed the required end-to-end delay bound and no losses;

ii) admitting the new flow does not make an established flow violate its required delay bound or lose packets.

In order to perform this test, a method for computing the worst-case delay that a flow experiences in a sink tree and the worst-case buffer occupancy at a node are required. Describing those methods is the subject of the next section.

3. Worst-case delay and buffer occupancy

---

\(^1\) It is also possible to define more than one sink tree rooted at an egress node, e.g. so as to insulate traffic with different QoS requirements.
in a sink tree

In this section we describe the formulas for computing the worst-case delay for a flow and the worst-case buffer occupancy at a node. Their complete derivation process is shown in [10], to which the interested reader is referred for the details. Let us first introduce two preliminary results:

**Theorem 1** ([10]): Consider a node \( x \). Let \( I \) be the set of ingress nodes of paths which include node \( x \), so that \( x = f_i(h_i) \) for some node \( i \in I \) and \( h_i, 1 \leq h_i \leq N_i \). Let \( \sigma_i, \rho_i \) be the leaky-bucket parameters for the flow entering node \( i \). Then, the aggregate flow at the output of node \( x \) is leaky-bucket shaped, with a burstiness \( s_x \) and a sustainable rate \( r_x \) as follows:

\[
s_x = \sum_{i \in I} \left[ \sigma_i + \rho_i \cdot \sum_{j \in j(k)} \theta_{f(i)} \right], \quad r_x = \sum_{i \in I} \rho_i \tag{1}
\]

and the values in (1) are tight output constraints at node \( x \).

Furthermore, a well known result regarding leaky buckets is the following:

**Proposition 2:** if two leaky-bucket shaped flows 1 and 2 are aggregated at a node, then their aggregate is still a leaky bucket shaped flow, with parameters \( \sigma_1 + \sigma_2, \rho_1 + \rho_2 \).

Let us now consider a sink tree as the one shown in Figure 1, and let us focus on a path \( P_i \). First of all, although an arbitrary number of flows can traverse that path, we do not need to distinguish them. In fact, by Proposition 2, we can describe their aggregate as a single leaky-bucket shaped flow. As far as delay bounds are concerned, in order for single flows to be guaranteed their respective required end-to-end delay bounds, the worst-case delay experienced by any bit traversing the path cannot exceed the minimum of those required delay bounds. Therefore – as far as the worst-case delay computation is concerned – we can assume without any loss of generality that one flow exists on a path, i.e. that we have one flow per ingress node. Accordingly, we denote with \( \sigma_i, \rho_i \) the leaky-bucket parameters of the flow traversing \( P_i \), and with \( \delta \) the required delay bound.

Based on Theorem 1 and Proposition 2, we can also model the aggregate traffic that joins path \( P_i \) at node \( f_i(h) \) (possibly entering it from different nodes) as a single flow. We call it the interfering flow \( (i,h) \), and we denote its leaky-bucket parameters as \( \sigma_{(i,h)}, \rho_{(i,h)} \). The following property shows how to compute the leaky-bucket parameters of an interfering flow from node parameters:

**Property 3.1:** Consider a path \( P_i \). Then, for \( 2 \leq h \leq N_i \) it is:

\[
\sigma_{(i,h)} = s_{f_i(h)} - \left[ s_{f_i(h-1)} + r_{f_i(h)} \cdot \theta_{f_i(h)} \right] \tag{2}
\]

Note that, in general, although for two paths \( P_i \) and \( P_j \), \( i \neq j \) it is \( f_i(h) = f_j(k) \), interfering flows \( (i,h) \) and \( (j,k) \) may not be the same, which accounts for the need of a pair of subscripts for denoting them. In fact, from Property 3.1, given a node \( x = f_i(h) = f_j(k) \), \( (i,h) \equiv (j,k) \) if and only if \( x \neq M_{i,j} \). For the network in Figure 1 (a portion of which is shown in more detail in Figure 2), it is \( M_{1,5} = f_3(2) = f_3(2) = 6 \), and \( (5,2) \neq (1,2) \); on the other hand, \( 10 = f_3(3) = f_3(3) \), and \( (1,3) \equiv (5,3) \). For symmetry of notation, we also define \( (i,1) \equiv i \), so that \( \sigma_{(i)} = \sigma_i \) and \( \rho_{(i,i)} = \rho_i \).

![Figure 2. Model for the path of the tagged flow](image)

Having said this, we now show how to compute the worst-case delay for a flow along a path. First of all, in order for queues not to build up indefinitely at a node \( x \), the following stability condition must be ensured:

\[
r_x \leq R_x \tag{4}
\]

And, if (4) holds for all nodes along path \( P_i \), the worst-case delay for the flow traversing that path is upper bounded by:

\[
V_i = \sum_{h=1}^{N_i} \left[ \theta_{f_i(h)} + \frac{\sigma_{(i,h)}}{CR_{f_i(h)}} \right] \tag{5}
\]

where \( CR_{f_i(h)} \) is the clearing rate at node \( f_i(h) \). The latter is the minimum rate at which a burst arriving at once at that node \( f_i(h) \) leaves the egress node. In general, \( CR_{f_i(h)} \) is a function of both the service rate \( R_{f_i(h)} \) and the sustainable rate of interfering flows \( \rho_{(i,h)} \) at nodes \( h \leq k \leq N_i \) along any path \( P_i \) that traverse node \( f_i(h) \). It can be computed once it is known which nodes act as bottlenecks for node \( f_i(h) \), according to the following definition.

**Definition 1:** consider two nodes \( x \) and \( y \), such that there is a path \( P_i \) that traverses them in that order, i.e. \( g_i(x) \leq g_i(y) \). Then, we say that \( y \) is a bottleneck for \( x \), and we write \( y > x \), if:

\[
R_y - r_y \leq \bigwedge_{x \leq i \leq y} \{ R_i - r_i \} \tag{6}
\]
Intuitively, node \( y \) is a bottleneck for node \( x \) if the amount of its overprovisioning \( R_x - r_x \) is smaller than (or equal to) the minimum overprovisioning at all nodes in the path from \( x \) to \( y \). Note that, by definition, it is \( x > y \). Let us call \( B_y = \{b'_y, b'_y, \ldots, b''_y\} \) the sequence of \( W_i \geq 1 \) bottleneck nodes for node \( x \), sorted in the same order as they appear in any path that traverses that node, so that \( b'_y = x \). Then, it is:

\[
CR_x = R_{b'_y} \cdot \prod_{i=1}^{W_y-1} \frac{R_{b'_i}}{R_{b'_i} + (r_{b'_i} - r_i)}
\]  

(7)

Note that we can also rewrite (7) equivalently as:

\[
CR_x = \begin{cases} 
\frac{R_{b'_y}}{R_{b'_y} + (r_{b'_y} - r_i)} \cdot CR_y & W_i > 1 \\
R_i & \text{otherwise}
\end{cases}
\]

(8)

which shows that the clearing rate at a node \( x \) can be computed recursively based on the clearing rate of the first downstream bottleneck node \( b'_y \), if there exists one. As far as worst-case delays are concerned, the following property can be easily proved.

**Property 3.2:** Consider any two paths \( P_i \) and \( P_j \), \( i \neq j \), and let \( x = f_i(h) = f_j(k) \). Then, it is:

\[
\sum_{m=h+1}^{N_i} \left[ \theta_{f(m)} + \frac{\sigma_{i,m}}{CR_{f(m)}} \right] = \sum_{m=k+1}^{N_j} \left[ \theta_{f(m)} + \frac{\sigma_{j,m}}{CR_{f(m)}} \right]
\]

(9)

Therefore, we can associate the summation in (9) to node \( x \) itself. Furthermore, since both sides of (9) are in fact the last addenda of \( V_i \) and \( V_j \), we call it the partial worst-case delay \( v_x \). Note that, by (2), it is:

\[
v_{f(h)} = v_{f(k)} + \theta_{f(h)} + s_{f(h)} - \frac{s_{f(h)} + r_{f(h)} \cdot \theta_{f(h)}}{CR_{f(h)}}
\]

(10)

for \( 1 \leq h < N_j \). Moreover, it is:

\[
v_i = v + \sigma - \frac{\sigma}{CR_i}
\]

(11)

As far as the worst-case buffer occupancy at a node \( x \) is concerned, a well-known Network Calculus result [11] proves that it is equal to the burstiness of the output flow at that node. Therefore, we can guarantee that no losses due to buffer overflow occur at node \( x \) if and only if:

\[
s_x \leq S_x
\]

(12)

In the next section, we describe the admission control algorithm and analyze its complexity.

### 4. Admission control algorithm

When a new candidate flow requests admission in the network along a path \( P_i \), it advertises its leaky-bucket parameters \( \sigma, \rho \) and its required delay bound \( \delta \). Accepting the candidate flow would imply that:

- the leaky-bucket parameters for the flow traversing path \( P_i \) are increased: \( \sigma \rightarrow \sigma + \sigma, \rho \rightarrow \rho + \rho \);
- the required delay bound for the flow traversing path \( P_i \) might decrease, in case the one of the candidate flow is smaller: \( \delta \rightarrow \delta \land \delta \).

As a consequence, the following three admission control tests should be performed:

1. Check if there is sufficient bandwidth and buffer to accept the candidate flow along the path, i.e. if (4) and (12) would still hold after admitting the flow.
2. Update the worst-case delay \( V_i \) for the flow traversing path \( P_i \), according to (5), and check if \( V_i \leq \delta_i \), also given that the latter might have decreased.
3. For flows traversing all other paths \( P_j \) in the sink tree, the worst-case delay \( V_j \) needs to be updated too, according to (5), so as to check if \( V_j \leq \delta_j \) still holds. In fact, given \( M_{i,j} = f_i(a), V_j \) depends on \( \sigma_{i(a)}^j \cdot P_{i(a)} \), which, by Theorem 1, depend in turn on \( \sigma_j, \rho_j \).

If all the three tests are successful, then the flow can be admitted. Note that the amount of computations required for the third test only depends on the number of paths, i.e. on the number of ingress nodes in the sink tree, rather than on the actual number of single flows established in the sink tree. In the rest of this section, we investigate how the amount of required computations depends on the network parameters.

We start with observing that, in order to compute the new worst-case delay for a path \( P_i \), we need \( N_i \) clearing rates, and therefore \( N_i \) sequences of bottleneck nodes \( B_{j(k)}, 1 \leq k < N_i \). However, since a node can be traversed by several paths, its sequence might have been already computed previously, and therefore we should not compute it twice. Furthermore, depending on the value of the network parameters, when a new flow is admitted we might not even need to compute it at all. Consider the example shown in Figure 3, in which three flows 1, 2 and 4 are established in the sink tree along paths \( P_1 = \{1,3,5\}, P_2 = \{2,3,5\}, P_3 = \{4,5\} \). From Theorem 1 we straightforwardly compute rates \( \rho_{(1,2)} = \rho_2, \rho_{(1,3)} = \rho_{(2,3)} = \rho_2, \rho_{(2,3)} = \rho_1, \rho_{(4,2)} = \rho_3, \rho_2 \). Accordingly, it is \( B_1 = \{1\}, B_2 = \{2\}, B_3 = \{3\}, B_4 = \{4,5\}, B_5 = \{5\} \).

Assume that a new flow with sustainable rate \( \rho \leq 20 \) be added at node 2. In that case, the bandwidth along path \( P_2 \) is sufficient for admitting the flow (we overlook the buffer occupancy for the sake of readability), so we should compute \( V_1, V_2, V_4 \) and check whether they are
acceptable. By applying the definition, we obtain that if \( \rho \geq 10 \) then \( B_i \rightarrow [1,3] \).

\[
\begin{align*}
\rho_1 = 25, & \quad R_1 = 20 \quad f_1(1) \\
\rho_2 = 35, & \quad R_2 = 35 \quad f_2(2) \\
\rho_3 = 50, & \quad R_3 = 50 \quad f_3(3) \\
\end{align*}
\]

Figure 3. Example of sink tree

On the other hand, if \( \rho < 10 \), then \( B_i \) remains the same. However, the other four sets cannot change regardless of the value of \( \rho \). In fact:

- A sequence of bottleneck nodes \( B_i \) depends only on the traffic joining the sink tree after node \( x \) (see (6)). Thus, \( B_1, B_2, B_3, \) and \( B_4 \) cannot change.

- The sequence \( B_4 \) includes all nodes in path \( P_4 \). Now, from (6), after the admittance of a new flow the rate of interfering flows at node 5 can only increase. Thus, node 5 will still be part of the sequence after the flow is admitted. So, \( B_5 \) cannot change.

In summary, a careful inspection of the network tells us that we would only need to update at most one set of bottleneck nodes (out of five) in order to reflect the change in the network parameters and be able to compute the new worst-case delays for all the flows through (5).

In the next subsections, we describe the data structures employed for modeling a sink tree, also showing the procedures needed when a flow is admitted and terminated. Furthermore, we evaluate the complexity required for updating those data structures, from which we derive an upper bound on the complexity of the admission control algorithm.

4.1. Data structures

We store information about the state of each path and node in the sink tree. For each path \( P_b \), we store the overall burstiness \( \sigma_b \) and sustainable rate \( \rho_b \) of the flows traversing it. Furthermore, for each single flow \( \alpha \) traversing that path, we need to store its own required delay bound \( \delta_{\alpha} \), so as to compute \( \delta_b = \min \{\delta_{\alpha}\} \), the required delay bound to be guaranteed along that path. Note that, when a new flow requests admission, in order to update \( \delta_b \), we would only need the smallest delay bound required so far by a flow established on the path (rather than storing all the required delay bounds for the single flows). However, we might need to update \( \delta_b \) also when a flow is terminated, if that flow had the minimum required delay bound, in which case all the delay bounds for flows traversing \( P_b \) would be necessary. Thus, we store the delay bounds of the flows in a min heap. Inserting and extracting elements of known value from a min heap takes \( O(\log d_b) \) operations, \( d_b \) being the number of different delay bound values (in the worst case, as many as the number of flows on path \( P_b \)). Furthermore, we store the array \( \{f_b(j), 1 \leq j \leq N_b\} \) of the label of each node in the path. We assume that initially (i.e. at network initialization) \( \sigma_0 = 0, \rho_0 = 0, \delta_i = \infty \), meaning that no flow has been established yet.

For each node \( x \), we store the rate-latency description \( R_x, \theta_x \) and the buffer space \( S_x \). Moreover, we store the leaky-bucket parameters of the output flow, \( (s_b, r_b) \), as computed by Theorem 1. Initially, it is \( s_b = 0 \) and \( r_b = 0 \). Furthermore, we store the clearing rate \( CR_x \). The latter can be efficiently computed through (8) once the label of the first downstream bottleneck node in \( B_x \), \( b_x^1 \), is known. Therefore, we also store \( b_x^1 \) (if \( B_x \) includes no other node than \( x \), we assume \( b_x^1 = \infty \)). Initially, no flow is active in the network, and therefore \( b_x^1 \) is set to the label \( y \) of the first downstream node whose rate \( R_y \) is not greater than \( R_x \) (or to \( \infty \), if there is no such node). Computing \( b_x^1 \) may take as many operations as the node’s depth, i.e. its distance from the egress node, and it is therefore the most critical operation. However, once we know \( CR_y \), \( CR_x \) can be computed in constant time by applying (8). We also store the partial worst-case delay \( v_j \) at that node, initialized to the sum of the latencies starting from the next downstream node up to the egress node. Finally, since a node can be traversed by more than one path, three flags are required in order to avoid updating a node’s data structures more than once, namely:

- \( upd_x \), which is set if node \( x \) has already been considered during the ongoing operation.
- \( newB_x \), \( newCR_x \) that are set if \( b_x^1 \), \( CR_x \) have been updated during the ongoing operation.

The three flags are false at all nodes, and they are set to true only while an update operation is being carried out.

Finally, we assume that the merge nodes \( M_{ij} \) are stored for each pair of paths \( P_i, P_j \). For instance, they can be stored in a symmetric matrix.

4.2. Admission of a new flow

In this subsection, we present the sequence of operations required to update the data structures when a new flow is admitted along path \( P_b \). We will show later in Section 4.3 that the operations to be performed when a flow is terminated are symmetric. Assume \( \{\sigma, \rho, \delta\} \) are the parameters for the flow. As a first step, the data structures related to path \( P_b \) need be updated. This includes
insertion of the delay bound $\delta$ in a min heap and computation of the new min value, i.e. $O(\log d_i)$ operations, plus two summations for updating $\sigma_i$ and $\rho_i$. As far as nodes in path $P_i$ are concerned, the following property shows that their first downstream bottleneck node cannot change.

**Property 4.1:** Let us consider a path $P_i$. Assume that the rate of the interfering flow $(j, h)$ changes. Then, for each node $y = f_j(n)$, $h \leq m \leq N_j$, $b^i_{y}$ does not change.

When a new flow joins path $P_i$, along that path only interfering flow $(i, 1)$ changes. Therefore, according to Property 4.1, no node along $P_i$ changes its first downstream bottleneck node. Furthermore, no other term in the clearing rate of nodes along path $P_i$ changes either (see (8)), and therefore their clearing rates need not be updated. Finally, the partial worst-case delay on each node in $P_i$ does not change either. Thus, the only information that need be updated in those nodes is $(s_i, r_i)$. This can be done by looping through all nodes in path $P_i$, as shown in Figure 4.

```
1. function new_flow (burst $\sigma$, rate $\rho$, path i) {
2.   int sum_theta=0
3.   for (h=1 to N_i) {
4.     r[f_i(h)]+=p
5.     sum_theta+= $\theta$[f_i(h)]
6.     s[f_i(h)]+= $\sigma$+p*sum_theta
7.     upd[f_i(h)]=true
8.   }
9. }
```

**Figure 4. Updating path $P_i$**

The number of operations to be performed in each iteration is constant. Thus, the loop completes in $O(N_i)$ steps. Note that the stability condition (buffer occupancy) at each node on the path can be tested after line 4 (line 6). Therefore, at the end of the loop, the first admission control test can be performed. As far as the second admission control test is concerned, the new value for $V_i$ can be computed in $O(1)$ time through (11).

After the loop has completed, the data structures of (potentially) all nodes in the sink tree that are not along path $P_i$ might need be updated. In fact, their first downstream bottleneck node might have changed, and their clearing rate might have too. However, we can limit the computations by taking into account both Property 4.1 and the following one.

**Property 4.2:** Let us consider a path $P_i$. Assume that the rate of the interfering flow $(j, h)$ changes. Then, for a node $y = f_j(n)$, $1 \leq n < h$, $b^i_{y}$ may change only if $g_j(b^i_{y}) > h$, or if $b^i_{y} = \infty$.

When updating $b^i_{y}$, we can capitalize on the following property that reduces the number of nodes that are candidate to become the new $b^i_{y}$.

**Property 4.3:** Let us consider a path $P_i$. Assume that the rate of the interfering flow $(j, h)$ increases. Then, for a node $x = f_j(n)$, $1 \leq n < h$, the new $b^i_{x}$, call it $b^i_{n, x}$, is such that:

- $g_j(b^i_{n, x}) \geq h$;
- $g_j(b^i_{n, x}) \leq g_j(b^i_{x})$, if $b^i_{x} \neq \infty$.

After $b^i_{x}$ has been updated for possibly all the nodes in case 3, the clearing rates of the nodes along path $P_i$ might need be updated. However, by (8) this is only required for a node $x$ if any of the conditions below is true:
- $b^i_{x}$ has changed;
- $CR_{b^i_{x}}$ has changed;
- the rate of one interfering flow $(j, k)$ such that $g_j(x) < k \leq g_j(b^i_{x})$ has changed.

Thus, when the rate of a flow entering path $P_i$ changes, it is enough to consider only the nodes along path $P_i$ from the ingress up to node $M_{i,j}$. In this case, condition c) translates to testing that $g_j(b^i_{x}) \geq g_j(M_{i,j})$, i.e. that $b^i_{x}$ is equal to $M_{i,j}$ or downstream with respect to $M_{i,j}$ on path $P_i$. Furthermore, for each node in path $P_i$ being considered, $s_i$ and $r_i$ do not change. Thus, after the data structures of nodes along path $P_i$ have been updated, those for nodes in a path $P_i \neq P_i$ are updated as shown in Figure 6.

**Figure 5. Possible cases for a path $P_i$**

Consider a node $x$ along path $P_i$. Based on the above properties, three cases are given, also shown in Figure 5.
- Node $x$ is after node $f_j(h)$ (and, obviously, so is $b^i_{x}$). In this case $b^i_{x}$ cannot change by Property 4.1.
- Both node $x$ and $b^i_{x}$ are before node $f_j(h)$. In this case $b^i_{x}$ cannot change by Property 4.2.
- Node $x$ is before node $f_j(h)$, while $b^i_{x}$ is after $f_j(h)$, therein including the case $b^i_{x} = \infty$. In this case $b^i_{x}$ may change. This is thus the only case in which updating $b^i_{x}$ might be required.
1. function update_path (path j, path i) {
2.     k=M[i,j]
3.     while (k>1) {
4.         if (upd[fj(k)]==true) k--
5.         else break
6.     }
7.     for (h= k down to 1) {
8.         update v[fj(h)] by applying (10)
9.     }
10.    } // update b2[fj(h)]
11.    if (gj[b2[fj(h)]]>gj[M[i,j]]) {
12.        y=first_dwn_bn(fj(h))
13.        if (y≠b2[fj(h)]) {
14.            b2[fj(h)]=y
15.            newB[fj(h)]=true
16.        }
17.    } // update CR[fj(h)]
18.    if (newB[fj(h)]==true) or (newCR[b2[fj(h)]]==true) or (gj[b2[fj(h)]]>=gj[M[i,j]]) {
19.        C=clearing_rate(fj(h))
20.        if (C≠CR[fj(h)]) {
21.            newCR[fj(h)]=true
22.            CR[fj(h)]=C
23.        }
24.    }
25.    upd[fj(h)]=true
26. }

Figure 6. Updating the nodes in path \( P_j \)

Within the while loop (lines 3-5) we identify the first node along path \( P_j \) that has not been updated yet. Starting from that node and going backwards (line 6-27), for each node \( x = f_j(h) \), we compute \( v_x \) (line 7) and we update \( b_x^2 \) (lines 8-15), by calling function \( \text{first_dwn_bn}() \) described below, and we update \( CR_x \) (lines 16-25) only if needed. Furthermore, whenever \( b_x^2 \) and \( CR_x \) are updated, we set the flags new\( B_x \) and new\( CR_x \). At the end of the pseudocode, \( V_j \) can be computed in \( O(1) \) operations through (11), and therefore condition \( V_j \leq \delta_j \) can be tested. Based on Property 4.3, function \( \text{first_dwn_bn}() \) can be written as in Figure 7 below.

1. function first_dwn_bn(node x, y) {
2.     j= any path traversing node x
3.     if (b2[x]==∞) last=Nj
4.     else last=gj(b2[x])
5.     for (k=gj(y) to last)
6.         if (R[fj(k)]-r[fj(k)]<=R[x]-r[x])
7.             return fj(k)
8.     return ∞
9. }

Figure 7. Updating \( b_x^2 \) - admission of a flow

The new \( b_x^2 \) can be computed in a number of operations which is linear with the depth of node \( x \) in the sink tree. Since in Figure 6 we consider the nodes backwards on a path, when node \( x \) is examined, either \( b_x^2 \) is \( ∞ \) or \( CR_x \) has already been computed. Thus, computing \( CR_x \) through (8) takes \( O(1) \) operations.

The update operation for the nodes not belonging to path \( P_i \) can also be formulated in a recursive way, as a preorder traversal of the sink tree starting from the root (egress) node. This makes the algorithm amenable to parallel implementation, possibly capitalizing on the capabilities of symmetric multi processor machines (SMP). The pseudocode for the recursive function is shown in Figure 8. Note, however, that the admission of different flows on a sink tree cannot be parallelized. In fact, the state of all the nodes need be updated before a new operation can be started.

1. function update_node(node x, path i) {
2.     j= any path traversing node x
3.     if (upd[x]==false) {
4.         update v[x] by applying (10)
5.         // update b2[x]
6.         if (gj[b2[x]]>gj[M[i,j]]) {
7.             y=first_dwn_bn(x,M[i,j])
8.             if (y≠b2[x]) {
9.                 b2[x]=y
10.                newB[x]=true
11.            }
12.        } // update CR[x]
13.        if (newB[x]==true) or (newCR[x]=true) or (gj[b2[x]]>=gj[M[i,j]]) {
14.            C=clearing_rate(x)
15.            if (C≠CR[x]) {
16.                newCR[x]=true
17.                CR[x]=C
18.            }
19.        }
20.        upd[x]=true
21.    }
22.    for_each y = child_of(x)
23.        update_node(y, i)
24. }

Figure 8. Recursive function for updating the nodes in the sink tree

Finally, when the update is completed, upd, new\( B_x \), and new\( CR_x \) are to be reset at all nodes, so that a new
update operation can be initiated. In summary, the operations required to update the data structures when a new flow is admitted are:

- inserting its required delay bound in a min heap at the ingress node and computing the new minimum $\delta_i$ ($O(\log d_i)$ operations);
- updating the data structures on the nodes along path $P_i$ ($O(N_i)$ operations) and performing the first two admission control tests ($O(1)$ more operations);
- updating the data structures on the nodes which are not along path $P_i$ (up to $O(N_j - g_i(x))$ operations for each node $x$ along path $P_i$);
- computing $V_j$ at each ingress node and performing the third admission control test ($O(1)$ more operations per ingress node once they have been updated);
- Resetting the flags on all nodes, which takes $O(N)$ operations, $N$ being the number of nodes in the sink tree.

Therefore, assuming that $D = \max \{ N_i \}$ is the depth of the sink tree, and $d = \max \{ d_i \}$ is the maximum number of different delay bound values at a single ingress node, the complexity of the update operation is $O(\log d + D \cdot N)$.

4.3. Termination of a flow

The operations required for updating the data structures at the termination of a flow are symmetric to those required at the admission. First of all, one extraction from the min heap at its ingress node is required, which has the same complexity as one insertion. Furthermore, as far as nodes are concerned, the termination of a flow can be regarded as the admission of a virtual flow with a “negative” burstiness and sustainable rate traversing the same path. Therefore, since Properties 4.1 and 4.2 still hold, the pseudocode shown in Figure 4 and Figure 6 (or Figure 8) can still be applied without any modification. However, Property 4.3 no longer holds in this case, and therefore we need to rewrite function $\text{first_dwn_bn}()$. We can formulate a property which is symmetric to Property 4.3:

Property 4.4: Let us consider a path $P_i$. Assume that the rate of the interfering flow $(j, h)$ decreases. Then, for a node $x = f_j(n), 1 \leq n < h$, the new $b^*_x$, call it $b^*_x^*$, is such that:

- $b^*_x^* = \infty$, if $b^*_x = \infty$;
- $g_f(b^*_x^*) \geq g_f(b^*_x)$, if $b^*_x \neq \infty$.

Based on Property 4.4, we rewrite function $\text{first_dwn_bn}()$ as in Figure 9. The new $b^*_x^*$ can still be computed in a number of operations which is linear with the depth of node $x$ in the sink tree. Therefore, the complexity involved in updating the data structures when a flow is terminated is the same. In summary, the optimum admission control algorithm has a complexity of $O(\log d + D \cdot N)$, which only depends on the number of different delay bound values at a single node and on the network topology.

```
1. function $\text{first_dwn_bn}$(node x, y) {
2.     if (b2[x] == \infty) return \infty
3.     else {
4.         j = any path traversing node x
5.         for (k=gj(b2[x]) to Nj)
6.             if (R[fj(k)] - r[fj(k)] <= R[x] - r[x])
7.                 return fj(k)
8.         return \infty
9.     }
10. }
11. }
```

Figure 9. Updating $b^*_x^*$ - termination of a flow

4.4. Implementation issues

As to how the admission control algorithm is to be implemented, two options are possible: centralized (on a domain basis) and distributed. A centralized entity, (e.g. a bandwidth broker), having full control of the whole domain, could in fact perform further optimizations by dynamically tuning the provisioning of shared resources (e.g., bandwidth on the physical links) among different sink trees, based on the traffic demand at the ingress. For scalability reasons, the algorithm could instead be implemented at the egress nodes, each one managing the subset of sink trees they are a root for and acting as part of a distributed bandwidth broker system.

As far as the state, i.e. the data on which the algorithm is performed, is concerned, we observe that it needs not be entirely stored on the entity that runs the algorithm. More specifically, the min heap of delay bounds could be stored directly at each ingress node. Ingress nodes could then communicate the new minimum delay bound to the centralized entity through signaling whenever needed. The entity that runs the algorithm needs to store two copies of the nodes data structures. In fact, a flow can be granted admission once the three admission control tests described at the beginning of Section 4 have been passed, and testing those conditions entails modifying the data structures as if the flow had been admitted. Thus, the updates are to be performed on a shadow copy of the data structures, and committed (e.g., by making the shadow copy actual) only after all the tests have been passed. In case the algorithm is implemented in software, this entails copying the node data structures (only few bytes per node). From a complexity standpoint, this boils down to adding a line in the pseudocode of Figure 4 and Figure 6 (or Figure 8), which does not affect the overall complexity of the algorithm.
Finally, we observe that, in a line of principle, an entirely distributed implementation of the proposed admission control algorithm could also be envisaged. This would entail distributing the node data structures on the node themselves, having some processing done on the nodes and employing an ad-hoc inter-node signaling in order to update them at every admission/termination of a flow. However, such a solution would require more intelligence to be put into the nodes, increase the amount of signaling traffic and probably make the flow setup latency considerably higher.

5. Related work

In this section we review the most relevant related work regarding delay-based admission control in sink-tree networks.

First of all, the problem of deriving per-flow delay bounds in networks employing aggregate scheduling has attracted a considerable amount of research in the last years (e.g., [3]-[17]). While a method for deriving tight bounds in generic network topologies has so far remained elusive [18], recent achievements show that Network Calculus is a promising tool at least for deriving bounds holding for specific topologies. For instance, a delay bound for a flow traversing a tandem of latency-rate servers, at each one of which it is multiplexed with leaky-bucket shaped traffic, has been derived in [19]. Furthermore, a delay bound holding for sink-tree topologies has been derived and proved to be tight in [10]. A method for deriving delay bounds in generic feed-forward networks employing aggregate scheduling has also been presented in [17], although no delay expression is reported therein. In [10], we prove that the method yields arbitrarily loose delay bounds when applied to sink-tree topologies.

In [3] an endpoint admission control system based on sink-tree resource management has been proposed. Only two classes of services are considered, real-time and best-effort, which are scheduled according to a strict priority paradigm. Authors propose four different schemes for sharing bandwidth among different routes and different classes of service. All the schemes rely on the computation of an end-to-end delay bound for a real-time flow, computed as the sum of the worst-case delays of nodes along its path. The latter, in turn, are computed differently based on the adopted sharing scheme. The admission control scheme proposed in [3] employs a worst-case delay formula that depends on resource allocation among different sink trees, on the topology and on the parameters of real-time flows. Hence whenever a real-time flow requires the admission in the network, the admission control algorithm need be run anew.

In [6] authors take into account the same problem dealt with in this paper, though in a slightly different context. They propose an admission control algorithm for sink trees of constant-rate nodes traversed by fluid leaky-bucket shaped flows with a finite peak rate. In our system model, the network is composed of variable-rate nodes, which generalizes the above settings to the case in which several independent aggregates share the physical link bandwidth. Here, we assume that the peak rate of leaky-bucket shaped flows be infinite. While this is certainly an approximation, it is reasonable in high-speed network environments. The algorithm proposed in [6] is based on an upper bound which is tight in additive sink-tree networks (i.e. those in which the maximum end-to-end delay is equal to the sum of the local maximum delays). It can be proved that a sink-tree network is additive if and only if, for each node \(x\), \(B_x = \{x\}\) (i.e., \(b'_x = \infty\)). In this particular case, the algorithm described in [6] is equivalent to the one proposed in this paper. In every other case, the delay bound used in [6] is not tight.

It is worth noting that the worst case delay formula on which the admission control algorithm proposed in this paper is based (5), along with Theorem 1 and Proposition 2, represent actual worst-case descriptions. Therefore the algorithm proposed in this paper rejects a flow only if there exists a combination of arrivals at ingress nodes (subject to the leaky-bucket constraints) and node behaviors (subject to the rate-latency guarantee) such that a bit of a flow in the sink tree actually either exceeds the required delay bound or overflows a node’s buffer. Thus, unlike the other proposed algorithms, the admission control algorithm which is based on those tests is optimum, i.e. achieves the maximum possible utilization, given the hypotheses of the system model.

6. Conclusions and future work

In this paper we have presented and analyzed an admission control algorithm for real-time traffic in sink-tree networks. The presented algorithm is based on a worst-case delay which has been derived and proved to be tight by using Network Calculus. The presented algorithm is optimum, i.e. it only rejects a flow if a chance actually exist that the required delay bound be exceeded, or losses due to buffer overflows occur in the network. Its complexity depends on the network topology (i.e., on the product of the depth by the number of nodes), and on the logarithm of the maximum number of flows aggregated at a single ingress node.

This work can be extended in many directions. The first one, already ongoing at the time of writing, is a thorough evaluation of the actual number of operations required for admitting a new flow (whereas in this paper we have evaluated the maximum number), and how this num-
ber is related to the resource provisioning. We expect sink trees with the same depth and number of nodes to exhibit different performance depending on (e.g.) the maximum node span, or how rates are provisioned along paths. Also, a more extensive evaluation of the utilization of the network resources that can be achieved under aggregate scheduling is required. Furthermore, the analysis shown so far assumes static resource provisioning, i.e. a given partition of link bandwidth and buffer among aggregates traversing the same node. Taking into account dynamic resource provisioning schemes that maximize the amount of traffic (or the total revenue associated to it) is therefore a possible evolution. As a last issue, classification criteria at the ingress nodes are also assumed to be given as an input. Researching on how to partition flow into aggregates so as to maximize the resource utilization is then a desirable goal.

7. Acknowledgements

This work was supported by the Italian Ministry of Education, Universities and Research (MIUR) in the framework of the FIRB-Perf project.

8. References


