A Novel Approach to Scalable CAC for Real-time Traffic in Sink-Tree Networks with Aggregate Scheduling

Luciano Lenzini  Linda Martorini  Enzo Mingozzi  Giovanni Stea
Dipartimento di Ingegneria dell’Informazione, University of Pisa, Italy
Via Diotisalvi, 2 56122 Pisa, Italy - Ph. +39 050 2217599
{l.lenzini, l.martorini, e.mingozzi, g.stea}@iet.unipi.it

ABSTRACT
In this paper we investigate the problem of scalable admission control for real-time traffic in sink-tree networks employing per-aggregate resource management policies, like MPLS or DiffServ. Every traffic flow entering the network at an ingress node, and flowing towards a given egress node, specifies its leaky-bucket parameters and a required delay bound for traversing the network. We propose an algorithm that admits a new flow if a guarantee can be given that the required delay bound, besides those of other already established flows, are not exceeded. We identify properties of sink-tree networks based on which we considerably reduce the complexity of the proposed algorithm, and we show that the latter approaches the theoretical lower bound on the worst case complexity of any algorithm working under the same hypotheses. Finally, we show that the algorithm lends itself to a distributed implementation, thus allowing for better scalability.

Categories and Subject Descriptors

General Terms
Algorithms, Performance, Design

Keywords
Admission Control, Network Calculus, Aggregate Scheduling.

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, 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, 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 multipoint-to-point Label Switched Paths, which also allows for saving label space compared to a point-to-point strategy, thus achieving better scalability. How to efficiently 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 for its sink tree and to the traffic at the ingress nodes. The worst-case delay can be computed for any utilization factor up to 100%, and it is tight, i.e. actually achievable. This makes this result a solid building block for devising efficient traffic management schemes suitable for real-time traffic in sink-tree networks. For instance, in [20] an admission control algorithm exploiting that formula has been described. The algorithm has been devised for implementation in a bandwidth broker. It checks if 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. Such admission control algorithm 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 other already established flow’s) traffic violate its required delay bound. However, as sink-tree resource management is proposed to enhance scalability, an admission control algorithm should be scalable itself, in order to not represent a bottleneck for the network performance. Therefore, in this paper, we focus on how to make such an algorithm scalable. More specifically, we identify the critical sub-task in the algorithm from the point of view of scalability, i.e. updating the data structures required to compute the end-to-end delay bounds along each path, and we present properties that allow one to considerably reduce the number of operations required to perform this sub-task. Furthermore, we show that the
resulting algorithm approaches the theoretical lower bound on the worst-case complexity of any algorithm working under the same hypotheses. Finally, we sketch guidelines for distributing the computation among ingress and egress nodes in order to further enhance scalability.

The remainder of the paper is organized as follows: Section 2 introduces the system model and states the problem. Section 3 describes the worst-case delay formula on which the algorithm is based. In Section 4 we describe the admission control algorithm, whose scalability is analyzed in Section 5. We review the related work in Section 6. Finally, we report conclusions in Section 7.

2. SYSTEM MODEL AND PROBLEM STATEMENT

We focus on an arbitrary topology network domain (e.g., a Diff-Serv or MPLS domain), modeled as a network of switching elements (e.g., IP routers) connected by links. Following standard practice, we define as node a network element which adds variable delay to traffic, e.g., 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 an egress node, is created at network setup time. For the sake of generality, we assume that every node in the tree, being either a leaf or an intermediate node, may be an ingress node of the network, thus acting as an ingress point of traffic.

- Traffic directed to an egress node is marked at each ingress as belonging to the sink tree rooted at that node (e.g., by assigning an ad-hoc DiffServ CodePoint 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, which is therefore treated as a single aggregate. As far as scheduling is concerned, the only constraint we put is that the service that each node guarantees to a queue can be modeled through a rate-latency service curve. Such a definition includes a large number of well-known schedulers, e.g. Packet Generalized Processor Sharing [14], and Deficit Round Robin [16], which guarantee a minimum departure rate.

Based on the above hypotheses, 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 sink tree, which we model as a network of $N$ variable-rate FIFO servers. $R_i$, $\theta_i$ and $S_i$ denote the guaranteed rate, latency and buffer space at node $i$, respectively, for that sink tree.

A path $P_i$ is a loop-free sequence of $N_i$ nodes, from an ingress node to the egress one. 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 position 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 say that the two paths merge at that node, i.e. $M_{ij} = f_i(a) = f_j(b)$, for some $a, b$ such that $1 \leq a \leq N_i$ and $1 \leq b \leq N_j$ and $f_i(a-1) \neq f_j(b-1)$. Without loss of generality, we assume the nodes are labeled so that each path is an increasing label sequence from the ingress node towards the egress one. It is worth noting that if two paths $P_i$ and $P_j$ merge at node $x = f_i(h) = f_j(k)$, they share all nodes from the node $x$ up to the egress node. Thus, we can use the ingress node label as a path subscript, i.e., $f_i(1) = i$, without any ambiguity. Fig. 1 shows a sink tree with 10 paths defined.

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

- that flow can be guaranteed the required end-to-end delay bound and no losses;
- admitting the new flow does not make an established flow violate its required delay bound or lose packets.

In order to perform this test, methods 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 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 $x = f_i(h)$ for some node $i \in I$ and $1 \leq h \leq N_i$. Let $\sigma_i, \rho_i$ be the leaky-bucket parameters for the fresh 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:

$$s_x = \sum_{i \in I} \sigma_i + \rho_i \cdot \sum_{h \in I \cap (h)} \theta_{f(h)}(h)$$

$$r_x = \sum_{i \in I} \rho_i$$

(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 $\sigma = \sigma_1 + \sigma_2 + \rho_1$.

Let us now consider a sink tree as the one shown in Fig. 1, and let us focus on a path $P_i$. First of all, although an arbitrary number of flows can traverse that path (i.e. enter at the same ingress node), we do not need to distinguish them. In fact, by Proposition 2, we can describe their aggregate at the ingress of the path as a single leaky-bucket shaped flow. As far as delay bounds are concerned, in order for each single flow to be guaranteed its required end-to-end delay bound, the worst-case delay experienced by any bit of the aggregate cannot exceed the minimum of the delay bounds required by each single flow. Therefore we can assume without any loss of generality that one flow traverses a path $P_i$, i.e. we have one fresh flow per ingress node. Accordingly, we denote with $\sigma_i, \rho_i$ the leaky-bucket parameters of that flow and with $\delta_i$ its 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:
\[
\sigma_{(i,h)} = s_{f_i(h)} - \left[ s_{f_i(h-i)} + r_{f_i(h-i)} + \theta_{f_i(h)} \right]
\]
\[
\rho_{(i,h)} = r_{f_i(h)} - r_{f_i(h-i)} \tag{3}
\]

Note that, in general, although for two different paths $P_i$ and $P_j$, $f_i(h) = f_j(k)$, interfering flows $(i, h)$ and $(j, k)$ may not be the same (hence we need a pair of subscripts for denoting them). In fact, from Property III.1, given a node $x = f_i(h) = f_j(k)$, $(i, h) = (j, k)$ if and only if there exist a node $y < x$ such that $y \in P_i, P_j$. In the network of Fig. 1 (a portion of which is shown in more detail in Fig. 2), we can see that paths $P_i$ and $P_j$ merge at node 6 = $f_i(2) = f_j(2)$ and $(1, 2) \neq (5, 2)$. Furthermore we define flow $(i, 1)$ as the sum of the output flows at all children of node $i$ (if there are any) and the fresh traffic entering node $i$. For instance, at a leaf node, $\sigma_{(i,0)} = \sigma_i$ and $\rho_{(i,0)} = \rho_i$.

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 to not build up indefinitely at a node $x$, the following stability condition must be ensured:
\[
r'_x = R_x - r_x \geq 0 \tag{4}
\]

Where $r'_x$ is called the residual rate of node $x$, i.e. the rate which is not strictly necessary to sustain the admitted traffic.

If (4) holds for all nodes along path $P_i$, the worst-case delay for the flow traversing that path is upper bounded by:
\[
V_x = \sum_{i=1}^{N_i} \left[ \theta_{f_i(h)} + \sigma_{(i,h)} / CR_{f_i(h)} \right] \tag{5}
\]

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.

**Fig. 2. Model for the path of the tagged flow**

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 $\rho_{f_i(h)}$ at nodes $h \leq k \leq N_i$. 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 $y \succ x$, if:
\[
r'_y \leq \min_{z \in P_i} \left\{ r'_z \right\} \tag{6}
\]

Intuitively, node $y$ is a bottleneck for node $x$ if its residual rate is the minimum among all nodes in the path from $x$ to $y$. Note that, by definition, it is $x \succ x$. Let us call $B_i = \{ b'_1, b'_2, \ldots, b'_i \}$ 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 $b'_i = x$. Then, it is:
\[
CR_x = \prod_{j=1}^{W_i} \frac{R_{b'_j}}{R_{b'_j} - r_{b'_j}} \tag{7}
\]

Note that we can also rewrite (7) equivalently as:
\[
CR_x = \begin{cases} 
\frac{R_x}{R_x - r_x} & \text{if } W_i > 1 \\
R_x & \text{otherwise}
\end{cases} \tag{8}
\]

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 $b'_1$, i.e. its first downstream node with a smaller or equal residual rate, if there exists one.

As far as worst-case delays are concerned, the following property can be easily proved.

**Property 3.2:** Consider two paths $P_i$ and $P_j$, $i \neq j$, and let $x = f_i(h) = f_j(k)$. Then, it is:
\[
\sum_{i=1}^{N_i} \left[ \theta_{f_i(h)} + \sigma_{(i,h)} / CR_{f_i(h)} \right] = \sum_{j=1}^{N_j} \left[ \theta_{f_j(k)} + \sigma_{(j,k)} / CR_{f_j(k)} \right] \tag{9}
\]

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 $V_x$ and $V_y$, we call it the partial worst-case delay $v_x$. Note that, by (2), it is:
\[ V_x(h) = V_{x(h+1)} + \frac{s_{x(h+1)} - s_{x(h)} + \theta_{x(h+1)} \cdot \rho_{x(h+1)}}{CR_{x(h+1)}} \]

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

\[ V_i = V_i + \sigma_{i(h)} / CR_i \]

It is worth noting that the worst-case delay computed using (11) is actually achievable (i.e., given a flow that traverses path \( P_i \), there exists a scenario in which one bit of that flow actually experiences a delay equal to \( V_i \)).

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:

\[ s_x \leq S_x \]

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 delay bound required by the candidate flow is smaller: \( \delta \rightarrow \delta + \delta \).

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 \( V_i \leq \delta \), also given that the right hand term might have been 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 \( M_{j,i} = f_j(x) \), \( V_j \) depends on \( \sigma_{j(a)} \cdot \rho_{j(a)} \), which, by Property III.1 and Theorem 1, depend in turn on \( \sigma, \rho \).

If all the three above tests are successful, then the candidate flow can be admitted. Note that any algorithm that is based on checking the above three conditions is optimum, because it only rejects a candidate flow if a chance actually exists that accepting it would make a bit of its (or some other already established flow’s) traffic violate its required delay bound. Moreover it allows both bandwidth and buffer to be utilized up to 100% (see (4) and (12)).

Besides testing if a candidate flow can be admitted, an admission control algorithm has to update the state of the network (e.g., decrease or increase the amount of resources available at each node) and to update the delay bounds for all paths whenever a flow is admitted or released.

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 \( B_{f_{i(k)}} \), \( 1 \leq k \leq N_i \). However, since a node is shared among 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 Fig. 3, in which three flows 1, 2 and 4 are established in the sink tree along paths \( P_1 = (1,3,5), P_2 = (2,3,5), P_4 = (4,5) \). From Theorem 1 we straightforwardly compute rates \( \rho_{1,2} = \rho_2, \rho_{1,3} = \rho_2, \rho_{2,3} = \rho_1, \rho_{4,5} = \rho_1 + \rho_2 \). Accordingly, it is \( B_1 = 1, B_2 = 2, B_3 = 3, B_4 = 4, B_5 = 5 \). Assume that a new flow with leaky-bucket 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 \( B_1 \rightarrow (1,3) \).

Fig. 3. Example of sink tree

On the other hand, if \( \rho < 10 \), then \( B_1 \) remains the same. However, the other four sequences of bottleneck nodes 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 admission (see (6)). Thus, \( B_2, B_3 \) and \( B_4 \) cannot change.
- The sequence \( B_i \) includes all nodes in path \( P_i \). 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 record the change in the network parameter and be able to apply (5) so as to compute the new worst-case delays for all the flows.

In the next subsections, we describe the procedure required for testing the admission of a flow and for updating the state of the network when an established flow is terminated. The data structures required for both procedures are introduced as they are used.

4.1 Admission Control Test

Assume that a flow characterized by \( \{\sigma, \rho, \delta\} \) requests to be admitted at ingress node \( i \). In order to test the three admission control conditions, we need to update the state of each node. For each node \( i \) in the sink tree we store both static information, i.e., those related to topology and resource provisioning (e.g., \( R_i, \theta_i \)), and dynamic information like \( \sigma_i, \rho_i, s_i, r_i, b_i, CR_i \) and \( V_i \). The latter can be modified when the number of admitted flows change, i.e., when a flow requests admission and when an established flow terminates. However, the admission control test is subject to failure, and therefore such information needs be stored in two separate copies: an actual copy, describing the actual state of the node, and a working copy, which describes the future state.
of the node assuming that the ongoing admission control test will eventually succeed. When a flow requests admission, the working copy of each node is updated based on the actual copy. If the admission control test succeeds, the working copy stores the correctly updated sink tree state, hence the two copies are swapped. Conversely, if the admission control test fails, then the working copy is simply discarded. In the pseudo-code that follows, we chose not to increase the notational burden to distinguish the two copies, since the context makes it clear.

First of all, for all the nodes \( x \) in path \( P \), the leaky bucket parameters of the output flow need to be updated according to Theorem 1. This can be done by visiting those nodes, as shown in Fig. 4.

![Fig. 4. Function that updates the nodes in path \( P \).](image)

During the above loop, the first admission control test is performed. Furthermore, it is easy to see that no node along \( P \) changes its first downstream bottleneck node (in fact, their residual rate is decreased of the same quantity \( \rho \)), and that no other term in the clearing term of rates of nodes along path \( P \) changes either (see (7)). Therefore, \( b^1_x \), \( CR_x \) and \( v_x \) need not be updated if \( x \in P \). However, after the above loop has completed, the data structures of (potentially) all nodes \( x \in P \) might need be updated. In fact, their first downstream bottleneck node, clearing rate and partial delay might have changed.

The most critical task is finding out, for a node \( x \), whether \( b^1_x \) has changed and, if so, which node is the new \( b^1_x \). The concept of bottleneck node is related to inequalities among residual rates of nodes along a path. In order to facilitate the computation of \( b^1_x \) after a new flow request, we introduce the new concept of slack node of a given node.

**Definition 2:** Consider two nodes \( x, y \in P \), such that \( x < y \). Then, we say that \( x \) is a slack node along path \( P \) of \( y \), and we write \( x = s_x(y) \), if:

\[
x = \{ w \in \tilde{P}_{x} : w < y, r_w < r^*_w \}
\]  

(13)

That is, node \( x \) is the nearest upstream node along path \( P \) such that \( r_w < r^*_w \). Node \( y \) can have at most one slack node for each path traversing it. We thus define the set of slack node of \( y \), \( SN(y) = \{ w : \exists P, w = s_x(y) \} \). However, we observe that it is expectable that \( |SN(y)| \) be smaller than the number of paths traversing node \( y \), since if \( x \in P_x \) and \( x = s_x(y) \), then \( x = s_x(y) \) as well.

Clearly, if \( r_w \) is reduced due to the arrival of a new flow, \( y \) might become first downstream bottleneck node for some of its upstream nodes in the sink tree. The following important property holds:

**Property 1.** Consider node \( y \in P_x \), and assume that a new flow with a rate \( \rho \) is admitted in path \( P \). Then, \( \forall x \in P_x \), \( x \not\in P \), a necessary condition for \( y \) to become \( b^1_x \) is that \( r_x^* - \rho \leq r_x^* \), where \( w = s_x(y) \).

**Proof:** see [22]

Thus, when a new flow traverses path \( P \), we can check if \( y \in P \) has become the new first downstream bottleneck node for other nodes by only comparing its residual rate with that of nodes in \( SN(y)/\{s_x(y)\} \). Let us then consider nodes \( y \) and \( x \in SN(y)/\{s_x(y)\} \). If \( r_x^* - \rho \leq r_x^* \), then, according to (13), \( x \) is no more included in \( SN(y) \), and \( y \) becomes \( b^1_x \). Furthermore, the following property can be proved:

**Property 2:** Consider node \( y \in P_x \) and \( x \in P_x \), \( x \not\in P \) such that \( x = s_x(y) \). Assume a new flow with rate \( \rho \) is admitted along path \( P \) and \( r_x^* - \rho \leq r_x^* \). Given \( z = s_z(x) \), either of the following two cases is given:

- If \( r_x^* - \rho > r_z^* \) then \( z = s_z(y) \) (i.e. \( z \) is the new slack node of \( y \) along path \( P \)).
- If \( r_x^* - \rho \leq r_z^* \) then \( y = b_x^1 \) (i.e. \( y \) becomes the new first downstream bottleneck node of \( z \)).

**Proof:** see [22]

If \( y \) becomes the first downstream bottleneck node for one of its slack nodes \( x \), then nodes in \( SN(y) \) either become slack nodes of \( y \) or will have \( y \) as a first downstream bottleneck node.

We can exploit the above two properties to devise a function that efficiently updates the first downstream bottleneck nodes in a sink tree, whose pseudocode is shown in Fig. 5.

First of all, when a new flow requests to be admitted at path \( P \), only nodes belonging to \( P \) are candidate to become first downstream bottleneck node for other nodes. Thus, we move backwards along path \( P \) and consider each node \( y \). We compare the new residual rate \( r_y \) to that of its slack nodes, considering them in decreasing label order, (lines 4-5). If it is no greater, we

- remove \( x \) from \( SN(y) \) and add \( SN(x) \) to \( SN(y) \) (line 6)
- update \( b^1_y \) (line 7).

Note that we do not know for sure whether a node \( w \in SN(x) \) will actually become a slack node for \( y \) (in fact, according to Property 2, it might be that \( y = b_x^1 \)). However, we add it to \( SN(y) \) and compare \( r_y \) against \( r_{w} \) at a later iteration of the loop. Finally, according to (8), if \( b_x^1 \) changes, we need to update \( CR_x \). Hence, we set a flag newFBN to remember this at a later stage (line 8).

![Fig. 5. Function that updates the first downstream bottleneck node (FBN).](image)

As an example, consider the sink tree shown in Fig. 6, in which \( r_0^* = 5, r_1^* = 8, r_2^* = 10, r_3^* = 12, r_4^* = 15 \).
Assume that a new flow with rate $\rho = 9$ is admitted at path $P_i$, so that $r_i \rightarrow 3$ and $r_i' \rightarrow 6$. We move upstream along path $P_i$, i.e., we start with the root node $4$. According to (13) $SN(4) = \{2, 3\}$, hence we compare $r_i'$ to $r_i$ (node 3 belongs to path $P_i$, hence it is not considered). Since $r_i' \leq r_i'$, we update data structures as follows: $b_2^i \rightarrow 4$ and $SN(4) \rightarrow SN(4) \setminus \{2\} \cup SN(2) = \{0, 1, 3\}$. Then we compare $r_i'$ to $r_i'$. Since $r_i' \leq r_i'$, we set $b_2^i \rightarrow 4$ and $SN(4) \rightarrow SN(4) \setminus \{1\} \cup SN(1) = \{0, 3\}$. Finally we compare $r_i'$ to $r_i'$. In this case $r_i' > r_i'$, hence no update is needed. We then move backwards along path $P_i$, and examine node 3. However, $SN(3) = \emptyset$, hence the update is complete.

Once $b_2^i$ has been updated, $CR_x$ might need be updated. Furthermore we need to update the partial delay $\delta_x$ for all nodes $x \notin P_i$.

The above two operations can easily be performed by visiting the sink tree from the root node to the leaves (since the clearing rate of a node depends on the clearing rate of downstream nodes). The recursive function that performs this task is shown in Fig. 8. As far as updating the clearing rate is concerned, by (8) this is required only if any of the conditions below are true (lines 3-5):

- $b_x^{i'}$ has changed (i.e. if $newFBN_x = true$)
- $CR_x$ has changed;
- given $j$ such that $x \in P_j$, the rate of one interfering flow $(j, k)$ such that $g_j(x) < k \leq g_j(b_x^{i'})$ has changed. If the new flow traverses path $P_j$, this condition is verified if, and only if, $g_j(x) < g_j(M_{i,j}) \leq g_j(b_x^{i'})$, as shown in Fig. 7.

If any of the above conditions are true, $CR_x$ might need be updated according to (8) (line 5-7), and the flag $newCR_x$ is set in order to test the second condition on upstream nodes.

Finally, we update the partial delay $\delta_x$ for all nodes $x \notin P_i$, according to (10) (line 8).

In summary, the operations required to test the admission of a new flow are shown in Fig. 9. First of all, all flags are reset (lines 2-3), and $\sigma_i$ and $\rho_i$ are increased (line 4). Then the nodes in path $P_i$ are updated (line 5), also checking the first admission control test, and the second test, i.e. whether $\delta_x \leq \min\{\delta, \delta_x\}$ (line 6-7), is performed. Then, for each node $x$, $b_x^{i'}$, $CR_x$ and $\delta_x$ are updated (lines 8-9), and the third admission control test is checked (lines 10-12). If all the three tests are successful, the working copy of the data structure is swapped with the actual copy (line 13). Finally, we need to update the required delay bounds for path $P_i$. In order to do this, we store the required delay bounds $\delta_{i,x}$ of each single flow $\alpha$, in a red-black tree [21] (line 14).

In the next subsection we show the algorithm for the release of an established flow.

**4.2 Release algorithm**

Assume that a flow, characterized by $\{\sigma, \rho, \delta\}$ and traversing path $P_i$, is released. The algorithm to be performed at the release of a flow is shown in Fig. 10. Note that, since a flow release is not subject to failure (as is a flow admission), we can update the actual copy of the data structures directly.

**4.2.1 Flow release**

At each node $x$, $b_x^{i'}$ is updated by the UpdateFBNRelease() function, which we describe in detail below. Let us define $FBN^{-1}(y) = \{x \in P_i | \alpha_y = b_x^{i'}\}$, the set of nodes whose first downstream bottleneck node is $y$. Consider a node $y \in P_i$ and $x \in FBN^{-1}(y), x \notin P_i$. For each $y \in P_i$, $r_i' \rightarrow r_i + \rho$. If
$r_i^* + \rho > r_j^*$, then $y \neq b_j^*$, and therefore $x$ has to be removed from $FBN^{-1}(y)$. Moreover it is possible to prove the following property, which is dual to Property 2:

**Property 3:** Consider node $y \in P_i, P_j, x \in P_i, x \notin P_j, x \in FBN^{-1}(y)$. Assume a flow with rate $\rho$, admitted at path $P_i$, is released and that $r_i^* + \rho > r_j^*$. Given $w = b_j^*$, two cases are given:

- If $r_i^* + \rho \leq r_j^*$, then $w = b_j^*$ (i.e. $w$ is the new first downstream bottleneck node of $x$)
- If $r_i^* + \rho > r_j^*$, then $x = s_x(w)$ (i.e. $x$ becomes the new slack node of $w$ along path $P_j$)

**Proof:** see [22].

We can exploit the above property to devise a function that efficiently updates the first downstream bottleneck nodes in all the nodes in the sink tree, shown in Fig. 11. When a flow traversing path $P_i$ is released, only nodes belonging to $P_i$ increase their residual rate. Thus, we move forward along path $P_i$ and consider each node $y$ (line 2). We compare the new residual rate $r_i^*$ to that of nodes in $FBN^{-1}(y)$, considering them in increasing label order, (lines 4). If $r_i^* > r_j^*$, we

- remove $x$ from $FBN^{-1}(y)$ and add it to $FBN^{-1}(b_j^*)$ (lines 6-7)
- update $b_j^*$, and set the corresponding flag (lines 8-9).

Furthermore, we need to update $SN(y)$. Now, $x$ is a candidate to become $s_x(y)$, since $r_i^* < r_j^*$. However, from (13) we cannot be sure until we check the same inequality all nodes $z$ in path $P_i$, such that $x < z < y$. Thus, in the algorithm we add $x$ to $SN(y)$, and we remove any child node of $x$ (line 10). Since nodes in $FBN^{-1}(y)$ are visited in increasing label order, this guarantees that $SN(y)$ is updated correctly.

As an example, consider again Fig. 6 with $r_i^* = 12$, $r_j^* = 16$, $r_i^* = 10$, $r_j^* = 4$, $r_j^* = 8$. Assume that a flow with $\rho = 5$ traversing path $P_i$ is released, so that $r_i^* \to 15$ and $r_j^* \to 13$. We move downstream along path $P_i$, and consider each node $y$ (line 2). According to (6), it is $FBN^{-1}(2) = \{0, 1\}$. Thus, we compare nodes 2 and 0 and see that $r_i^* > r_j^*$, which implies that: $FBN^{-1}(2) = \{1\}$, $b_2^* = b_1^* = 4$. Then, we compare $r_i^* > r_j^*$. In this case, $r_i^* \leq r_j^*$, so no update is needed.

After checking all the nodes in $FBN^{-1}(2)$, we move downstream along the path to node 4 ($FBN^{-1}(4) = \{0\}$). We compare nodes 4 and 0 obtaining $r_i^* > r_j^*$, hence $b_4^* = b_0^* = \infty$ (i.e. after the releasing, node 0 has no first downstream bottleneck node).

1. function UpdateFBNRelease (path i)
2. for h = 1 to $N_i$
3. $y = P_i(h)$
4. for each node $x$ in $FBN(y)$, in increasing label order
5. if ($R_i - r_i > R_j - r_j$)
6. $FBN(y) = FBN(y) \cup \{x\}$
7. $FBN(b_2[y]) = FBN(b_2[y]) \cup \{x\}$
8. $b_2[x] = b_2[y]$  
9. newFBN = true
10. $SN(y) = SN(y) \cup \{x\}/children(x)$

**Fig. 11.** Function that updates the first downstream bottleneck node (FBN) after the release of a flow

In the next section, we assess the scalability of the proposed admission control algorithm: we compute the worst-case complexity of each function, and we sketch guidelines for distributing the computations and the state among nodes in a domain.

### 5. Scalability Assessment

Call $N$ the number of nodes in the sink tree, $D$ the tree depth, and $d$ the maximum number of different delay bounds requested at a single ingress node (which are no more than the number of admitted flows). We observe that:

- The $UpdatePath()$ function is $\Theta(D)$, since it loops through all the nodes in a path and it takes a constant number of operation per iteration;
- The $UpdateTree()$ function is $\Theta(N)$, since all nodes in the tree are visited once and each operation in the pseudocode can be implemented in constant time;
- resetting flags at all nodes is $\Theta(N)$;
- computing the new worst-case delays at all ingress nodes is $\Theta(N)$, since it takes a constant number of operation per ingress node;
- inserting (and extracting) elements from a red-black tree is $\Theta(\log d)$;
- swapping two sets of data structures can be done in constant time using pointers.

We now move to examining the complexity of the $UpdateFBNArrive$ and $UpdateFBNRelease$ functions. First of all, bit vectors can be used to represent $SN(y)$, $FBN^{-1}(y)$, and the set of children of a node, so that union and extraction require a constant number of operations. Furthermore, locating the $p^{th}$ largest (smallest) element in such a set can be done with a constant number of machine instructions. Therefore, each iteration in the inner loop of both functions requires a constant number of operations. Hereafter, we compute the number of iterations to be performed in the worst case for both function.

We start from $UpdateFBNArrive$. The function examines all the nodes $y \in P$, and compare $r_i^*$ to $r_j^*$, $x \in SN(y)$. If $r_i^* \geq r_j^*$, $SN(y)$ is modified as follows: $SN(y) = SN(y) \setminus \{x\} \cup \{x\}$. In the worst case, each node $z \notin P$, and belonging to the sub-tree that has node $y$ as root can be part of $SN(y)$. For the sink tree shown in Fig. 12, assume that the new flow traverses path $P_i$. In the worst case, $SN(11)$ can include, at different epochs, nodes 1,2,3,7 and 8.

![Fig. 12. A sink tree.](image)

Call $n(k)$ the number of nodes in the sub-tree that has node $k$ as root. The maximum number of iterations $C$ can be computed as follows:

$$C = \sum_{n=0}^{N-1} [n(f_i(n)) - (n + 1)]$$

that can be rewritten as follows.

$$C = \sum_{n=1}^{N-1} n(f_i(n)) - (N_i + 2)(N_i - 1)/2$$

Let us now consider the $UpdateFBNRelease$ function. When an established flow traversing path $P_i$ is released, we examine
again nodes \( y \in P_i \). For each node \( y \in P_i \) we compare \( r_y^* \) to \( r_y^r \), \( x \in FBN^+(y) \). If \( r_y^r < r_y^* \), then \( b_y^i \) is set to \( b_y^i \). The number of iterations is maximized when, for each node \( x \notin P_i, x \in P_j \), during time \( b_i^j \) is updated once for each node \( w \in P_i, P_j \). Consider again Fig. 12, and assume that the established flow traverses path \( P_i \). In the worst case \( b_i^j \) is set, in different iterations, first to 11 and then to 14. Hence, in this case, the maximum number of iterations \( C \) can be computed as follows:

\[
C = \sum_{i=1}^{N_i} \left[ \left( n(f_i(n)) - n(f_i(n-1)) - 1 \right) \cdot (N_i - n) \right]
\]

that, after simple algebraic manipulations, yields (15) again. Hence the maximum number of iterations required is the same in both cases.

It is worth noting that for a complete \( k \)-order tree with depth \( D \) we can write

\[
N = \sum_{i=0}^{D-1} k^i = \frac{k^D - 1}{k - 1}
\]

hence, in this case the maximum number of iterations is \( \Theta(N) \).

We observe that updating the first downstream bottleneck nodes after a flow is admitted or released is \( \Omega(N) \) in the worst case. In fact, it requires to re-sort a list of \( N \) residual rates after a subset of \( N_i \) elements are increased (or decreased) of the same quantity \( \rho \). This is equivalent to merging two sorted lists, one of \( P_i \) elements, and the other of \( N_i \) elements, which is known to be \( \Theta(N) \) in the worst case. This means that an admission control algorithm based on (5) is \( \Omega(N\log D) \) in the worst case. Therefore, our algorithm approaches the lower bound for the worst-case complexity on complete trees.

The admission control algorithm can be implemented on a centralized entity, such as a bandwidth broker, managing a whole domain partitioned in sink trees. However, it lends itself to distributed implementation. In fact, since sink trees are independent of each other, different instances of the algorithm can be run independently for each sink tree. Therefore, different instances of the algorithm can be run at egress nodes, each one managing the sink trees it is a root of, thus acting as part of a distributed bandwidth broker system. This would distribute the computational burden of admission control, without introducing additional communication overhead.

Furthermore, the computations can be further partitioned among egress and ingress nodes. In fact, the computation of the path delay bound, which entails maintaining the red-black tree and computing the new minimum delay bound along the path when the set of admitted flows change, can be performed directly at ingress nodes. The latter can then communicate the new minimum delay bound to the centralized entity at each admission control request or flow termination. Thus, each ingress node would only manage the subset of flows entering the domain through itself, partitioned into as many red-black trees as are the sink trees it is a node of.

6. RELATED WORK

In this section we review the most relevant related work regarding delay-based admission control in sink-tree networks. In doing this, we limit ourselves to considering works based on deterministic, worst-case traffic characterization and delay guarantees. Parallel streams of research, related to admission control algorithms relying on stochastic traffic models and/or measurement, are outside the scope of this work.

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 class 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 is 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., \( \delta_x^i = \infty \)). 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, along with Theorem 1 and Proposition 2, provides an achievable delay. 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. it achieves the maximum possible utilization, given the hypotheses of the system model. Another optimum algorithm, based on the same formula, has been derived in [20]. However, the latter does not exploit the concept of slack node introduced in Section 4.1, and thus makes redundant computations. More specifically, the first downstream bottleneck node for a node \( x \) is updated by comparing \( r'_k \) with \( r''_k \) for each node \( y \in P_x \). For instance, for the sink tree shown in Fig. 12, updating \( b^2_k \) would entail comparing \( r'_k \) with \( r''_k \), \( r''_{13} \) and \( r''_{14} \) in a worst case, regardless of the path traversed by the new flow (or by the flow to be released). According to the algorithm proposed in this paper, instead, updating \( b^2_k \) would entail comparing \( r'_k \) with \( r''_{14} \) alone. This implies that the maximum number of iterations performed according to the algorithm shown in [20] is in any case (i.e. for any topology and path traversed by a flow) larger than the one shown in (15). In fact, it can be computed as follows:

\[
C' = \sum n(x) \cdot (N_x - 1) 
\]

For instance, in a complete \( k \)-order tree with depth \( D \) : (15) becomes:

\[
C = \frac{k \cdot (k^D - 1)}{(k-1)} = \left[ \frac{k}{k-1} \cdot (D-1) \cdot \left( \frac{1}{k-1} + \frac{D}{2} \right) \right] 
\]

whereas (18) yields:

\[
C' = \frac{k}{(k-1)} \cdot \left[ 1 - (D+1)k^D + Dk^{(D+1)} \right] - D \cdot k^D 
\]

In Fig. 13 the difference \( C' - C \) is shown as a function of \( k \) and \( D \). As expected, the difference is always positive and grows quickly with both variables.

In Fig. 14 the percentage reduction \( \frac{(C' - C)}{C'} \) is also shown as a function of \( k \) and \( D \). We can see that, by exploiting the admission control algorithm presented above, the maximum number of iterations is heavily reduced compared with the maximum number of iterations proposed in [20]. It is worth noting that, when \( D \) increases, \( \frac{(C' - C)}{C'} \) grows and tends to become more and more independent of \( k \) as \( k \) increases. Furthermore, the relative gain of the proposed algorithm, over the one in [20], increases when both \( D \) and \( k \) increase, which proves that it better scales with respect to the size of the tree.

7. CONCLUSIONS

In this paper we have presented and analyzed an admission control algorithm for real-time traffic in sink-tree networks. The 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. We have derived new properties of sink-tree networks which allow to considerably reduce the number of operations required to test the admission of a flow. Furthermore, we have shown that the algorithm approaches the theoretical lower bound for the worst-case complexity of an optimum admission control algorithm, the latter being that of a merge-sort function. The algorithm lends itself to distributed implementation, in that i) different sink trees are independent of each other, and therefore can run different instances of the algorithm, and ii) the computations can be partitioned among an egress node and its ingresses.

This work marks the first step in the design of a bandwidth broker architecture suitable for real time traffic management in aggregate scheduling networks. In order to achieve this goal, several other issues need be investigated: for instance, a signaling protocol is required to transport requests and responses between the ingress nodes and the bandwidth broker managing the sink tree. We are currently implementing a prototype of the algorithm in order to carry out an accurate performance analysis and to evaluate its scalability. Finally a thorough evaluation of the utilization that can be achieved while providing deterministic worst-case guarantees is required.

8. ACKNOWLEDGEMENTS

This work has been supported by the Italian Ministry of Education, Universities and Research (MIUR) in the framework of the FIRB-Perf and QUASAR projects.
9. REFERENCES


