Efficient Link Scheduling for Online Admission Control of Real-time Traffic in Wireless Mesh Networks

P. Cappanera¹, L. Lenzini², A. Lori², G. Stea², G. Vaglini²

¹ Dip. di Sistemi e Informatica, University of Florence
Via S. Marta 3, 50139 Firenze – Italy
paola.cappanera@unifi.it

² Dip. di Ingegneria dell’Informazione, University of Pisa
Via Diotisalvi 2, 56122 Pisa – Italy
{l.lenzini, alessandro.lori, g.stea, g.vaglini}@iet.unipi.it

Abstract

Link scheduling is used in Wireless Mesh Networks (WMNs) to guarantee interference-free transmission on the shared wireless medium in a Time Division Multiple Access approach. Several papers in the literature address the problem of link scheduling guaranteeing a minimum throughput to the flows traversing the WMN. However, none of the existing works address the problem of computing a schedule that guarantees that pre-specified end-to-end delay constraints are met. In this paper, we make a first step forward in this direction by defining a link scheduling algorithm that works in sink-tree WMNs, i.e. those whose traffic is routed towards a common sink (i.e., the Internet gateway). Our iterative algorithm exploits a delay-based admission control procedure, devised through Network Calculus, which solves an optimization problem and tests the feasibility of a schedule from the point of view of delay guarantees. Thanks to a clever solution approach for the optimization problem, the iterative algorithm computes feasible solutions in affordable times for networks of several tens of nodes, and is thus amenable to online admission control of real-time traffic.

Keywords—Link Scheduling, Wireless Mesh Networks, Network Calculus, Sink-Tree Networks

1. Introduction

Wireless Mesh Networks (WMNs) are an emerging class of networks, usually built on fixed nodes that are interconnected via wireless links to form a multi-hop network. Their main goal is to provide broadband access to mobile clients who are just on the edge of wired networks. WMNs can be used where cable deployment is not feasible or is too expensive, such as in remote valleys or rural areas, but also in offices and home environments. End-users are served by nodes called mesh routers, which are generally assumed to be stationary. Mesh routers are in turn wirelessly interconnected so as to form a network backhaul, where radio resource management challenges come into play. Moreover, some mesh routers are generally provided with access (e.g. through wires) to the Internet and therefore can act as gateways for the entire WMN. Communication between any two mesh routers as well as from any router to gateways is multi-hop. Many of the WMN issues are thus common to multi-hop wireless networks. However, the fact that mesh routers are fixed makes the backhaul of a WMN inherently different from distributed wireless networks (e.g. ad hoc networks). For example, problems such as energy consumption are no longer an issue. This makes it sensible to opt for a centralized network management, as opposed to the distributed approaches used for ad hoc wireless networks. In this case, nodes act in a coordinated fashion under the supervision of a network entity which determines the management based on global knowledge of the network topology and additional conditions. The radio communication channel employed by WMNs (as by any other wireless networks) is broadcasting; i.e. a packet sent out by a mesh router will be received by all mesh routers tuned on the same frequency as the transmitter and within its transmission range, and furthermore it may cause signal interference to some mesh routers that are not intended to be the receivers.

One of the most widely used techniques to achieve robust and collision-free communication is link scheduling, operating in the context of Time Division Multiple Access (TDMA, [16]) where the time is slotted and synchronized.

¹ A preliminary version of this paper has appeared as [1].
Cross-layer approaches where, for example, the link scheduling and routing functionalities are jointly addressed have been extensively studied [2]–[6] in the past few years due to their application to TDMA MAC protocols. However, to the best of our knowledge, very few works published so far have addressed the problem of computing a link schedule with maximum end-to-end delays as a constraint. This is instead the objective of this paper, which is aimed at finding a conflict-free link scheduling such that the delay bounds of all its flows are not violated. Therefore, the paper findings widen the applicability field of a WMN to the class of real-time applications such as voice, video streaming, vehicular traffic control [25]–[26], video surveillance systems [27]–[28] or mission critical, i.e. applications which require a firm guarantee (i.e. an upper bound) on their maximum end-to-end delay. In this paper, sink-tree TDMA WMNs are analyzed, i.e. networks in which the root is an Internet gateway and for which routing is not an issue. The rationale behind this choice is that we want to focus on link scheduling and its impact on the end-to-end delay upper bound disregarding (at least for the moment) the routing functionality. We consider leaky-bucket-shaped flows traversing a sink-tree WMN which get aggregated as soon as they proceed towards the gateway. For these, the maximum end-to-end delay at the flow level (despite the aggregation) can be computed using a Network Calculus approach [7]. We first show that the link scheduling problem in these settings is mixed integer/nonlinear non differentiable, and, as such, very hard to solve in practice. Thus, we propose an alternative strategy to solve the same problem. We design an iterative algorithm: we compute an initial link scheduling, solving an affordable integer/linear optimization problem, that guarantees that the schedule is conflict-free. The schedule is then submitted to a Delay-Based Admission Control (DBAC) procedure [8], to test whether all the flows are within the required bounds. If the answer is negative, we refine the link scheduling problem, capitalizing on the feedback given by the admission control procedure (i.e., which flow violates its delay bound, and by how much), and we run another iteration. The algorithm terminates in a finite number of iterations, either computing a conflict-free schedule which is also feasible from a delay point of view, or failing to compute a conflict-free schedule under the given constraints.

In a highly dynamic environment, the flows can change over short periods of time. When a new flow joins the network, an admission control test has to be performed to test whether the network is able to support the same delay guarantees. Such a test involves computing a new link schedule according to the above-mentioned algorithm, and this has to be done in a reasonable amount of time (i.e., in the order of tens or hundreds of milliseconds). We therefore propose a clever heuristic solution for the integer/linear problem, that makes our algorithm amenable to online admission control for networks of up to several tens of nodes. Our heuristic allows effectiveness to be traded for computation time, but a favorable trade-off point can be reached where near-optimal results can be achieved in affordable times.

A recent work [17] deals with a similar problem, although in different settings. More specifically, it models a WMN as a stop-and-go system and formulates a min-max problem on the round-trip TDMA delay introduced by the scheduling. On the one hand, minimizing the maximum TDMA delay is a different problem with respect to computing delay-constrained link schedules. On the other hand, the model in [17] does not account for bursty traffic, hence neglecting queuing delays, which often represent the largest delay component (e.g., for compressed video traffic, which is inherently bursty). Other works ([18]–[19]) have already applied Network Calculus to wireless (sensor) networks in order to derive delay bounds. These works, however, derive delay bounds for a given network configuration. On the contrary, the scope of this paper is somewhat dual with respect to that, i.e. to configure the network (specifically, its link scheduling) based on delay bound constraints.

The rest of the paper is organized as follows: Section 2 reports the system model. The method through which we compute delay bounds is described in detail in Section 3. In Section 4 we formulate the delay-aware scheduling problem, and we present a heuristic approach to solve the latter in Sections 5 and 6. Section 7 reports a performance analysis. We review the related work in Section 8, and we draw conclusions and highlight directions for future work in Section 9.

2. System model

The framework developed in this paper relies on basic Network Calculus [9]–[12] concepts, i.e. those of arrival curve, service curve and delay bound. Interested readers can find the necessary background in [9], from which we also borrow some notation. The symbols used in this paper are summarized in Table 4 in the Appendix.

In our WMN it is assumed that each node (mesh router) is equipped with a single time-slotted channel. Transmission slots of a fixed duration $T_s$ are grouped into a frame of $N$ slots, which is periodically repeated every $N \cdot T_s$ time units. Each slot is assigned to sets of non-interfering links through conflict-free link scheduling, so as to prevent the wireless links from interfering with each other. Stated differently, during every slot, a subset of links may be activated for transmission with the assurance that no conflicts occur at the intended receivers. Therefore, a link $e$ which is activated for $\Delta_e$ slots in a frame starting from an offset $\pi_e$, can be characterized by means of a long-term minimum guaranteed rate equal to $R_e = c_e \cdot \Delta_e / N$, $c_e$ being its capacity, and by a vacation, i.e., the time interval between the end of an activation of $e$ and the beginning of the next one), equal to $\pi_e = (N - \Delta_e) \cdot T_s$. As such, it can be modeled as a rate-latency service curve [9], whose rate is the minimum guaranteed rate and whose latency is the vacation. Fig. 1 reports a graphic representation of the relevant quantities. We assume that a FIFO service discipline is in place at each link,
meaning that traffic from different flows is queued First-Come-First-Served. This means that bursts from one flow add to the queuing delay (and, hence, to the end-to-end delay) of others which traverse the same path. We assume that each link queue has enough buffer not to lose packets due to buffer overflows. In Section 3 we show how to dimension buffers so that this is guaranteed.

**Fig. 1 - Relevant quantities in link scheduling**

We assume that our WMN has a sink-tree (or multipoint to point) topology, as shown in Fig. 2, where flows entering a generic node travel towards the root node. The latter is possibly connected to a wired infrastructure, serving as a gateway to the Internet. In this paper we do not take into account the presence of downlink traffic which, as stated at the end of Section 3, is not a limitation of the proposed framework. In such a network, a path \( P_i \) is a loop-free sequence of \( n_i \) nodes, from an ingress node to the egress one. Given the tree structure of the network and the fact that a mesh router is equipped with a single channel, the node label can be used to address both the node and its output link without ambiguity. In order to denote a node’s position in a path, we define function \( l_i(h) \) that returns the label of the \( h^{th} \) node in path \( P_i \), \( 1 \leq h \leq n_i \), and function \( p_i(z) \) that returns the position of node \( z \) along path \( P_i \), i.e. \( p_i(z) = l_i^{-1}(z) \). Given two paths \( P_i \) and \( P_j \), \( i \neq j \), their traffic is aggregated at the first common node \( M_{i,j} \). We say that the two paths merge at that node, i.e. \( M_{i,j} = l_i(a) = l_j(b) \), for some \( a,b \) such that \( 1 \leq a \leq n_i \) and \( 1 \leq b \leq n_j \) and \( l_i(a-1) \neq l_j(b-1) \). 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 \( x = l_i(h) = l_j(k) \), 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. \( l_i(1:i) \), without any ambiguity. Fig. 2 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 shaped flow, with a burst \( \sigma \) and a sustainable rate \( \rho \). We assume that traffic is fluid, leaving packetization issues for further study. An admission control test for real-time traffic in such a network should let in a new flow if, and only if, that flow can be guaranteed the required end-to-end delay bound, and admitting the new flow does not make the established flows violate the required bounds. In order to perform this test, a method for computing the worst-case delay that a flow experiences in a sink tree is required. Describing that method is the subject of the next section.

**Fig. 2 - Paths in a sink tree**

3. **Worst-case delay in a sink-tree network**

In this section we describe the formulas for computing the worst-case delay for a flow in a sink-tree network of FIFO rate-latency nodes. Their complete derivation process is shown in [7], to which the interested reader is referred for the details. Let us first introduce two preliminary results:

**Theorem 1 ([7]):** Consider a node \( x \). Let \( I \) be the set of ingress nodes of paths which include node \( x \), so that \( x = l_i(h) \) for each node \( i \in I \) and \( 1 \leq h \leq n_i \). Let \( \sigma, \rho \) 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 burst \( s_x \) and a sustainable rate \( r_x \), as follows:

\[
s_x = \sum_{i \in I} \left[ \sigma + \rho \cdot \sum_{h=1}^{n_i} \theta_{i,h} \right] r_x = \sum_{i \in I} \sigma_i \rho_i \] (1)

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

The above result can be used for dimensioning the buffers at the nodes. In fact, as proved in [7], \( s_x \) is exactly the dimension required for avoiding buffer overflows. Furthermore, a well-known result regarding leaky buckets is the following:

**Property 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 \( \sigma + \sigma_i, \rho + \rho_i \).

Let us now consider a sink tree as the one shown in Fig. 2, and let us focus on a path \( P \). First of all, although an arbitrary number of flows can traverse that path (i.e. enter at the same node), we do not need to distinguish them. In fact, by Property 2, we can describe their aggregate at the ingress of the path as a single leaky-bucket shaped flow. In order to guarantee each single flow 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.e. we have one fresh flow per node. Accordingly, we denote with \( \sigma, \rho \) the leaky-bucket parameters of that flow and with \( \delta \) its required delay bound. If no fresh flow is injected at node \( i \), we can assume that a “null flow”, with \( \sigma = 0, \rho = 0, \delta = +\infty \), is injected in the network at that node.

Based on Theorem 1 and Property 2, we can also model the aggregate traffic that joins path \( P \) at node \( l_i(1:i) \), com-
posed of both arriving from upstream nodes and fresh flow injected at node \( l_i(h) \) itself, as a single flow. We call it the interfering flow \( (i,h) \), and we denote its leaky-bucket parameters as \( \sigma_{(i,h)} \) and \( \rho_{(i,h)} \). The following property shows how to compute the leaky-bucket parameters of an interfering flow from node parameters:

**Property 3:** In a path \( P_e \), for \( 2 \leq h \leq n_e \), it is:

\[
\sigma_{(i,h)} = s_{(i,h)} - \left[ s_{(i,h-1)} + r_{(i,h)} \cdot B_{(i,h)} \right], \quad \rho_{(i,h)} = r_{(i,h-1)} - r_{(i,h)}.
\]

Note that, in general, although for two different paths \( P_i \) and \( P_j \), \( l_i(h) = l_j(h) \), 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 3, given a node \( x = l_i(h) = l_j(k) \), \( (i,h) = (j,k) \) if and only if there exist a node \( y < x \) such that \( y \in P_i, P_j \). In the network of Fig. 2 (a portion of which is shown in more detail in Fig. 3), we can see that paths \( P_X \) and \( P_Y \) merge at node \( 5 = l_i(2) = l_j(2) \) and \( (0.2) = (3, 2) \) (both being easily identifiable through color codes in the figure). 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 not to build up indefinitely at a node \( x \), the following stability condition must be ensured:

\[
r_i = R_i - r_i \geq 0,
\]

where \( r_i \) is called the residual rate of node \( x \), i.e. the rate which is not strictly necessary to sustain the admitted traffic. If \( (2) \) holds for all nodes along path \( P_e \), the worst-case delay for the flow traversing that path is upper bounded by:

\[
D_e = \sum_{i=1}^{n_e} \left[ \theta_{(i)} + \sigma_{(i,0)} \cdot CR_{(i)} \right],
\]

where \( CR_{(i)} \) is the clearing rate at node \( l_i(h) \). The latter is the minimum rate at which a burst arriving at once at that node leaves the egress node.

**Fig. 3 - Relevant quantities for paths \( P_0 \) and \( P_3 \)**

In general, \( CR_{(i)} \) is a function of both the service rate \( R_{(i)} \) and the sustainable rate of interfering flows \( \rho_{(i)} \) at nodes \( h \leq k \leq n_i \). It can be computed once it is known which nodes act as bottlenecks for node \( l_i(h) \), according to the following definition.

**Definition 4:** Consider two nodes \( x \) and \( y \), such that path \( P \) traverses them in that order, i.e. \( p_i(x) \leq p_i(y) \). Then, we say that \( y \) is a bottleneck for \( x \) if:

\[
r_i^* \leq \min \left\{ r_j^* \cdot p_i(x) \leq p_i(y) \right\}.
\]

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, \( x \) is a bottleneck for itself. Call \( B = \{b_0^*, b_1^*, ..., b_n^*\} \) the sequence of bottlenecks for node \( x \), sorted in the same order as they appear in any path that traverses that node, so that \( b_0^* = x \). Then, it is:

\[
CR_x = R_{b_0^*} \prod_{y=1}^{w_x} \frac{R_{b_y^*}}{R_{b_y^*} + \left( R_{b_y^*} - r_y^* \right)}.
\]

Note that we can also rewrite (5) equivalently as:

\[
CR_x = \begin{cases} R_i + \left( R_i - r_i \right) \cdot CR_{b_i^*}, & W_x > 1, \\ R_i, & \text{otherwise} \end{cases}
\]

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 \( b_i^* \), i.e. the nearest downstream node with a smaller or equal residual rate, if there exists one. It is worth noting that the worst-case delay computed using (3) is actually achievable (i.e. given a flow that traverses path \( P \), there exists a scenario where one bit of that flow actually experiences a delay equal to \( D_e \)). However, it is a nonlinear and non-differentiable function of the nodes rates. In fact, non differentiability arises from the min operator in (4).

As a last note, we observe that, in this paper, we do not take into account the presence of downlink traffic. However, as far as delay bound computation is concerned, the above framework holds provided that uplink and downlink traffics are buffered separately, which can easily be achieved in today’s mesh routers.
4. Delay-aware link scheduling

In this section we formulate the delay-aware link scheduling problem. Our objective is to find a conflict-free link scheduling such that the delay bounds of all the network flows are not violated. The WMN is modeled through a connectivity graph, \( G = (V, E) \), where \( V = \{v_1, \cdots, v_n\} \) is a set of nodes representing the mesh routers and \( E = \{e_1, \cdots, e_m\} \) is a set of directed links that connect nodes in the wireless transmission range of each other. We first identify the constraints that ensure the conflict-free property, and then move to describing those related to delay feasibility.

We model the physical interference phenomenon occurring between the links of the wireless network by means of the widely used protocol interference models [13]-[15]. For each edge of the network \( e \in E \) we define a conflicting set of edges \( I(e) \) which includes all the edges belonging to \( E \) which interfere with \( e \) (\( I(e) \) contains \( e \) itself); the interference condition is straightforwardly defined as follows:

\[
\sum_{e \in I(i)} x_i(t) \leq 1, \text{if link } e \text{ is active in slot } t = 1, 2, \ldots, N,
\]

where \( x_i(t) \) is a binary variable, such that \( x_i(t) = 1 \) if link \( e \in E \) is active in slot \( t \), and 0 otherwise. This means that if edge \( e \) is active in slot \( t \), the associated interfering set \( I(e) \) must contain one active edge only (which is the edge \( e \) itself). We translate the interference condition to a conflict graph \( G_c = (E, C) \), where \( E \) is the set of links of the connectivity graph and \( C = \{e_1, \cdots, e_n\} \) is the set of edges that model conflicts within the network.

**Fig. 4 - Conflicts in a sink-tree TDMA network**

Many types of conflicts can be modeled using this class of interference models, but in a TDMA [16] network only a few types of conflicts are relevant [17]. In this work we focus on an upstream scheduling case, where only direct neighbor conflicts, i.e. only those conflicts between links that share the same upstream node, have to be considered. Half-duplex constraints need not be added to this model, as they are implicitly accounted for into the interference constraints, links being unidirectional. For instance, in the network topology shown in Fig. 4, left, for nodes 7 and 8 sharing the same upstream node, i.e. 12, we have the following edges in the conflict graph (shown on the right): \( (7,12), (8,12), (7,8) \).

Hence for each link \( e \) of the network, the set \( I(e) \) can be easily obtained by retrieving the one-hop neighborhood of node \( e \) in the conflict graph. In the example of Fig. 4 we have \( I(8) = \{2,3,7,8,12\} \). Given a conflict graph \( C \), only those conflicts between active links have to be considered. An active link is one with a non-null flow to be scheduled. We thus define \( C_i \subset C \) as the subset of conflicts involving active links:

\[
C_i = \{(i,j) \in C : f_i > 0 \text{ and } f_j > 0\},
\]

where \( f_i \) denotes the flow going through link \( i \).

In a similar way as in [17], we describe a schedule of length \( N \) using variables \( \pi_i \) and \( \Delta_i \), representing respectively the activation time of link \( i \) and the duration of its transmission (see again Fig. 1). Since time is slotted, such variables are non negative integers. Furthermore, for a schedule to be valid, each link must accomplish its transmission within the frame duration, i.e.:

\[
\pi_i + \Delta_i \leq N \quad \forall i \in E. \tag{6}
\]

The schedule must also ensure that the conflict-free condition is satisfied: while a link is transmitting, all of its conflicting links must refrain from transmitting. For any pair of active links \( i \) and \( j \) connected by an edge in the conflict graph we have:

- if \( j \) transmits after \( i \), it must wait for \( i \) to complete the transmission, i.e. the following must hold:
  \[ \pi_i - \pi_j + \Delta_i \leq 0 \]

- Otherwise, the symmetric inequality holds:
  \[ \pi_j - \pi_i + \Delta_j \leq 0 \]

In order to linearize the combination of the above constraints, we introduce a binary variable \( o_{ij} \), \((i,j) \in C_i \), called conflict orientation, which is 1 if \( i \) transmits after \( j \), 0 otherwise. The left-hand side of the previous constraints can therefore be upper bounded by \( N \) regardless of the relative transmission order, since \( \pi_i \) and \( \Delta_i \) vary between 0 and \( N \). This completes the formulation of what we will refer to as the conflict-free constraints, which are necessary and sufficient conditions:

\[
\begin{align*}
\pi_i - \pi_j + \Delta_i & \leq N \cdot o_{ij} \quad \forall (i,j) \in C_i \\
\pi_j - \pi_i + \Delta_j & \leq N \cdot (1 - o_{ij}) \quad \forall (i,j) \in C_j
\end{align*} \tag{7}
\]

Hereafter, we denote with \( S \) the feasible region of a conflict free schedule, i.e. the set of variables that satisfy con-
straints (6) and (7). Conflict-free constraints are independent of the traffic traversing the WMN. Additional constraints are needed to keep into account the end-to-end delay requirements of the various flows. We recall that parameters $\sigma_e$, $\rho_e$ and $\delta_e$ are respectively the burstiness, the sustainable rate and the required delay bound of the fresh flow joining the network at node $e$. Given such parameters, the end-to-end delay feasibility problem (E2EFP) is thus the following:

$$\begin{align*}
\text{find} & \quad \pi_e, \Delta_e, o_e \quad \forall e \in E \\
\text{s.t.:} & \quad D_e \leq \delta_e \quad \forall e \in E \\
& \quad \theta_e = (N - \Delta_e) \cdot T_e \quad \forall e \in E \\
& \quad R_e = c_e \cdot \Delta_e / N \quad \forall e \in E \\
& \quad \pi_e, \Delta_e, o_e \in S \quad \forall e \in E
\end{align*}$$

where $D_e$ is the delay bound defined in (3). Problem (8) is a feasibility problem, in that we want to find a solution, if there exists one. This problem is very hard to solve even for trivial instances, due to the fact that it is simultaneously integer, non linear and non differentiable (due to the delay bound expression), as discussed in Section 3. Hence, we design a heuristic iterative solution approach, which is described in the next section.

5. Iterative solution approach

Along its path to the gateway, each flow accumulates a delay bounded by (3). Such delay bound results from two components: the queuing delay and the scheduling delay. The former is due to both the bursty nature of the flow’s traffic and the traffic of other flows that share a part of the route, while the latter is due to the time it takes for the outbound link of a node to be scheduled within a frame. The queuing delay component is a strongly non linear expression in the variables of the problem. For this reason, solving the E2EFP problem directly is not viable. Instead, we design a heuristic solution algorithm, whose outline is shown in Fig. 5, that works as follows. First of all, we formulate a simpler integer linear problem which takes into account only the scheduling delay component, i.e. the latency at each link. More specifically, we formulate and solve a link scheduling problem whose objective function – to be minimized – is the weighted sum of the latencies at each link. We call this problem Minimum Weighted Latency (MinWL). The solution to the above problem may or may not verify the non linear delay constraints $D_e \leq \delta_e$. Therefore, we need to check whether the above inequality holds for all flows through a Delay-based Admission Control (DBAC), originally proposed in [8]. The latter tests whether the delay bound for all flows is below the required threshold. When this is not the case, it returns the indexes of the flows which violate their delay bound constraint, along with the maximum delay they would experience given the current link schedule. This information is then used to formulate another instance of the MinWL problem, obvious different from the former, with the objective of steering its solution towards the feasible region of the E2EFP. The algorithm stops when one of the following conditions occurs: a feasible solution is found, or the problem is declared unfeasible, or a generic pre-defined stopping criterion is met. As for stopping criteria, the one that should be employed in a real network is computation time: unless a feasible schedule is found by a predefined time frame (or the schedule is found to be unfeasible), then the flow is rejected.

![Fig. 5 - Outline of the heuristic approach](image)

The pseudocode for the iterative algorithm is shown in Fig. 6. An instance of the MinWL problem is solved (line 3). If a feasible solution is found, the corresponding schedule is analyzed using the DBAC procedure. This allows one to compute the maximum delay bounds associated to the current schedule (line 5) and to verify whether all the flows meet their deadlines. If some do not, a violating flow and one of its bottlenecks are selected. This information is then used to formulate another instance of the MinWL problem (line 11).

Clearly, the cornerstone of the above iterative algorithm are the MinWL scheduling problem and the heuristic feedback, which are thus described in detail the following subsections. For completeness, we also outline the DBAC, and we refer the interested reader to [8] for details regarding the data structures and algorithms involved.

![Fig. 6 - Pseudocode for the iterative algorithm](image)

5.1. Minimum weighted latency scheduling

Let us recall that, through (1), $r_e$ is the aggregate flow on link $e$. Being $c_e$ the capacity of each link $e \in E$, we force each link to transmit for a fraction of the frame no smaller than its utilization, expressed in term of the aggregate flow routed on link $e$ and the capacity of the link itself:
\[ \Delta_e \geq N \cdot r_e / c_e \quad \forall e \in E. \quad (9) \]

Constraint (9) leaves a degree of freedom to the solver, as it allows activations to be larger than strictly required for queues to be stable. Such extra allocation is in fact used to reduce the delays.

The delay bound of a flow along its path depends on the link latencies. The latter are linear functions of the scheduling variables \( \Delta_e \), for those links. For a link \( e \in E \) the latency is \( (N - \Delta_e) \cdot T_e \) and, as explained in Section 2, it represents the time interval between the end of an activation and the beginning of the subsequent one for link \( e \). Given a graph \( G \) and an assignment for \( \Delta_e \) and \( N \), we define the weighted latency of a network as follows:

\[ \text{WL}(G, \Delta) = \sum_{e \in E} w_e \cdot (N - \Delta_e) \cdot T_e. \]

The link weight \( w_e \) is set to the aggregate flow of link \( e \) itself, i.e. \( w_e = r_e \quad \forall e \in E \). The rationale behind this choice is that a link traversed by a large amount of aggregate flow has a high probability to become one of the network bottlenecks, hence the need to give it a larger weight. Therefore we write the MinWL scheduling problem as:

\[
\begin{align*}
\text{min} & \quad \text{WL}(G, \Delta) \\
\text{s.t.} & \quad \Delta_e \geq N r_e / c_e \quad \forall e \in E \\
& \quad \pi_e, \Delta_e, o_e \in S
\end{align*}
\]

This is clearly a linear integer program. Its worst-case complexity using a generic branch and bound approach is exponential in the number of integer variables (i.e., in the number of nodes and conflicts). Nonetheless, it can be optimally solved with general purpose solvers for instances of tens of nodes in a reasonable time (i.e., in the order of tens of seconds or minutes).

For homogeneous traffic demands (i.e., leaky bucket parameters) and regular tree topologies the MinWL problem alone appears to be a good approximation of the E2EFP [21], but the more heterogeneous the flows become, the less such an approximation is accurate. Therefore, we need an iterative structure with an incremental reformulation of the integer linear program, as explained in the following sub-section.

5.2. Delay-Based Admission Control

A feasible link schedule should satisfy the delay bounds for all the flows. A solution of the MinWL problem simply may not, as it optimizes on latencies, rather than having delay bounds as constraints. Hence we need to test a posteriori whether the delay bounds are actually within the requirements. This entails computing \( D_e \) through (3) for all the flows \( e \in E \), and checking whether \( D_e \leq \delta_e \). Note that a solution of the MinWL, by definition, verifies (2) because of the first constraint in (10), hence all the delays are finite at least.

In [8], we tackled a similar problem for a wired sink-tree network, and devised a solution that can be adapted to this task. In that work we present an algorithm that decides whether, given an admissible set of flows, admitting a new one would still be lead to feasible delays. The problem that we need to solve here is slightly different, meaning that all flows have to be considered at the same time. However, thanks to properties of sink-tree networks described in [8], a delay-based admission control can be performed at a small cost with respect to the other tasks involved in an iteration. In fact:

- \( \theta_e, R_e, s_e, r_e, r_e' \) can be computed at each node \( x \) with a single post-order visit of the tree. In fact, \( s_e \) and \( r_e \) only depend on quantities at children nodes through (1).
- \( \sigma_{[a]}(\rho_{[a]}) \) can be computed in constant time from the above, through Property 3, whenever required.
- For each path \( P \), the set of bottleneck nodes \( B = \{b_1, b_2, ..., b_n\} \) has to be computed. Because of (4), this can be done in at most \( n \) operations, hence it requires \( O(|E| \cdot H) \) operations, where \( E \) is the set of edges and \( H \) is the maximum depth of the tree.
- Computing (3) requires a sum of up to \( H \) terms per path, hence it requires \( O(|E| \cdot H) \) operations for the whole network.

Therefore, the cost of the DBAC procedure is \( O(|E| \cdot H) \), i.e. linear in the number of links. We remark that the above test is necessary and sufficient under the hypotheses of the system model, since (3) computes the actual worst-case delay. Hence, if \( D_e > \delta_e \) there actually exists a configuration of arrivals that makes one bit of flow \( e \) violate its required delay bound.

5.3. Heuristic feedback

A schedule given by the solution of an instance of the MinWL problem corresponds to assigning the link rates. The Network Calculus framework introduced in Section 3 can therefore be exploited to compute the flow delay bounds. One can easily check (and it is also formally proven in [7]) that the delay bound of a flow can be reduced by increasing the rate of *any* link along its path. However, bottlenecks play a special role, in that they have a smaller residual rate than
other links, i.e. they are accountable for a larger fraction of the delay accumulated traversing that path.

Accordingly, the heuristic feedback consists in reformulating the MinWL problem forcing a solver to give a higher rate to the bottlenecks of those flows that violate their deadline. More specifically, at each iteration the violating flow with the maximum difference between the actual and the required delay bound is selected:

\[
e = \arg \max_{i \in \gamma(j)} \{D_i - \delta_i\}
\]

where \(\gamma(i)\) is an indicator function, which is equal to 1 if flow \(i\) violates its delay bound and 0 otherwise. Its first downstream bottleneck is then given extra rate. This is done by substituting the constraints (9) with:

\[
\Delta_e \geq N(a_e \cdot K + r_e)/c_e \quad \forall e \in E,
\]

where \(a_e\) is the number of extra units of rate \(K\) to be scheduled for link \(e\). Variable \(a_e\) is initially null, and it is increased by one on each iteration for link (11). Constant \(K\) determines the granularity of the feedback: a smaller \(K\) allows a more fine-grained rate redistribution among the links, but also entails a slower convergence to a feasible solution (i.e., one where no flow violates its delay bound). Note that computing (11) too requires \(O(|E|)\) operations in a worst case.

In a line of principle, alternative strategies might be considered in the feedback step, both in the flow and the bottleneck selection: the last bottleneck could be used instead of the first one; furthermore, working on more than one flow at the same time is also possible, even if this option neglects the commonality between subpaths shared by two or more flows of the network. Nevertheless, all the above alternative strategies are outperformed by the one that we exploit in our solver. The iterative method proceeds with successive approximated formulations. While it cannot be guaranteed, in general, that each \(\Delta_e\) variable has a monotonically increasing trend over the iterations (and, accordingly, that the delay bound for each flow is decreasing), the alert reader can easily recognize that (12) represent increasingly constraining inequalities over the iterations due to the feedback component. This means that, as the algorithm iterates, either a feasible solution is found or some such constraints (12) will eventually be violated, leading to declaring the MinWL problem infeasible. Therefore, the iterative algorithm always terminates. We observe, however, that the maximum number of iterations could – in principle – be as large as the available network bandwidth measured in units of feedback \(K\), i.e. \(O(\sum_i c_i / K)\). However, note that – since our aim is to devise a link scheduling algorithm that can be used for online admission control of real-time traffic, the search for a feasible solution can be stopped when a given response time limit is exceeded.

From what has been shown so far, it is clear that – as far as efficiency is concerned – the critical block is the MinWL problem. In fact, most of the time of each iteration is spent in solving the latter, whereas both the DBAC and the feedback component are considerably less complex. More specifically, the complexity of a single iteration is exponential. In order to improve the efficiency of the algorithm, an efficient solution strategy for the MinWL problem is thus required. Before presenting our approach to solving the above problem, we give a comprehensive example of our solution algorithm by showing how it solves a simple instance of the E2EFP problem.

**Example**

Given a binary tree of 15 nodes, such as the one shown in Fig. 7, we want to schedule 14 flows originating from all the nodes of the sink tree. The leaky-bucket profiles for each flow are reported in Table 1, and the required delay bound is equal to 20. Parameter \(K\) is set to 500 units, while each link capacity is 9600. The frame duration is \(N\) is set to 100 slots2.

Table 2 reports the end-to-end delay bounds for each flow computed at each iteration of the MinWL problem: column labels \(I_0 - I_4\) are numbered according to the iteration of the algorithm to which they refer, and flows whose required delay bound is violated are highlighted. For instance, the solution of the initial instance \(I_0\) violates the delay constraints of flows 0 and 1.

Focusing on the bounds obtained at the first iteration \(I_0\), the flow with the maximum difference between the actual and the requested bound is flow 6 and its first downstream bottleneck is link 11. The extra rate assigned to link 11 at the subsequent iteration is sufficient to push flow 6 and flow 7 back within their required delay bound. However, in the new schedule flow 2 is now violating its deadline. Accordingly, its first bottleneck (i.e., link 9) is addressed. Note that it takes two iterations (\(I_1\) and \(I_2\)) for flow 2 to be pushed back within its deadline. At the third iteration (\(I_3\)) we have a maximum violation at flow 0, and we address bottleneck link 8. Note that the same link is also the first bottleneck for flow 1, which is also violating is deadline. Both flows are back in line after the feedback at \(I_3\). At iteration \(I_4\), flow 6 is again the one to target, and this time link 6 is its first bottleneck. After this iteration, we find a feasible solution for the E2EFP.

2 Note that here, and in the remainder of the paper, we intentionally choose to omit the units for the quantities (i.e., capacities, etc.), as our solutions do not depend on a specific technology, as long as the model described in Section 2 is adhered to.
6. Efficient approximate solution for the MinWL problem

In this section we take over the problem of devising an efficient solution algorithm for the MinWL problem. The presence of integer ($\pi_\ell, \Delta_\ell$) and binary ($o_j$) variables makes the MinWL complex. The optimal solution for integer-linear problems can be found in seconds to minutes for a single instance of a few tens of nodes (up to 50-60), depending on both the size of the network and the load. Larger instances sometimes cannot be solved at all, since the memory requirements associated with branch and bound operations (required by the integer and binary variables) quickly become too demanding. More to the point, in our approach, the MinWL problem needs to be solved several times for computing a single link schedule, due to the iterative structure. Therefore, in order to make such a framework feasible for online admission control purposes in networks of reasonable size, we need to devise a fast heuristic to solve the MinWL problem.

A first observation is that, by relaxing the integrality for $\pi_\ell, \Delta_\ell$ and assigning $o_j$ a value, we obtain a continuous linear problem, which can be solved in polynomial time (i.e., reasonably fast), [24]. Capitalizing on this, we devise a solution algorithm which is composed of two blocks (Fig. 8):

1. a first block that assigns values to each conflict orientation $o_j$ using a customized dive-and-fix heuristic [23];
2. a second block that solves a reduced MinWL problem, which emerges from the previous step once the conflict orientations are set, with relaxed integrality constraints. Its solutions are then rounded preserving the conflict-free property.

Fig. 8 – Improved-efficiency iterative solution scheme

Hereafter, we describe the above two blocks, and then we show how to modify the iterative algorithm of Fig. 5 in order to reap the full benefits of this solution in terms of efficiency.

A dive-and-fix heuristic exploits the information obtained from the linear relaxation of a mixed integer-linear problem (MILP) to fix all or some of its integer variables to integer values. It iteratively does the following: it solves (i.e., “dives” into) a linear relaxation of the MILP, it identifies a subset of integer variables to target (i.e., to “fix”), and rounds them to the closest integer. As variables are fixed, smaller MILPs are obtained for subsequent iterations. The procedure terminates when either all integer variables are fixed or a linear relaxation is detected to be infeasible. In a general context, when an infeasible relaxation is found, other techniques might be attempted (e.g., backtracking or local searches) before the original problem is declared infeasible.

We use a dive-and-fix approach to assign the $o_j$ variables (which are binary, hence integer) in the MinWL problem. Note that the latter also has other integer variables (i.e., $\pi_\ell, \Delta_\ell$), which are however not interested by the dive-and-fix procedure. They are instead considered later on for efficiency reasons.

With reference to the pseudocode in Fig. 9, we repeat the same procedure until all the $o_j$ variables are assigned. On each iteration, a linear relaxation of (a progressively reduced version of) the MinWL is solved. Then, we take the $o_j$ values obtained from the linear relaxation of the MinWL, and we compute the distance of each one to the closest integer (either 0 or 1). Then, (lines 7-13) all variables which are sufficiently close to an integer, i.e. that lie in an $\varepsilon$ - neighborhood of their closest integer are rounded, i.e. fixed for the subsequent iterations. As several variables often end up falling into reasonable tolerance factors (i.e., $\varepsilon = 0.01$), this considerably speeds up the process. Finally (lines 14-20), we also fix the one remaining $o_j$ having the minimum distance to its closest integer, large as the latter may be (if the distance is 0.5, a random Bernoullian fixing is made, with $p=0.5$).

The dive-and-fix iterates $O(\sqrt{C_j})$ times, at each step solving a linear problem. The latter has a solution time which is polynomial with the length of the binary encoding of the input (for a detailed discussion on its complexity please refer to [24]). Note that, since our emphasis is on efficiency, if the dive-and-fix returns an infeasible solution, we do not attempt time-consuming backtracking procedures or local searches, and instead declare the E2EFP problem infeasible. It is worth noting that, despite having been widely used in the last 15 years, the dive-and-fix heuristic has not been proved to approximate the optimal solution to a given degree, to present day. It is not even guaranteed that it can actually find a feasible solution if one exists. However, as we show later on, for this particular problem it performs rather well.

After fixing the conflict orientations, the second block comes into place. Each pair of constraints in (7) can now be
replaced by either of the following:
\[
\begin{cases}
\pi_i + \Delta_i \leq \pi_j & \text{if } o_{ij} = 0 \\
\pi_j + \Delta_j \leq \pi_i & \text{otherwise}
\end{cases}
\quad \forall (i,j) \in C_f (13)
\]

Hence, we substitute (13) for (7), obtaining a reduced MinWL, and solve the following linear relaxation of the latter:
\[
\begin{align*}
\min \quad & WL(G,\Delta) \\
\text{s.t.} \quad & \Delta_e \geq \lceil N((r_e + a_e \cdot K)/c_e) \rceil \quad \forall e \in E \\
& \pi_e, \Delta_e \in \tilde{S} \quad \forall e \in E 
\end{align*}
\quad (14)
\]

where \( \tilde{S} \) is the (continuous) feasible region given by \( \text{(6)} \) and \( \text{(13)} \). The solution values of this reduced model \( (\tilde{\pi}, \tilde{\Delta}) \), which are not necessarily integer, are truncated to their integer part. Note that the ceiling function is required in the first constraint in \( \text{(14)} \) to prevent the rounding from reducing the minimum guaranteed rate below the required one.

The truncated solution is still feasible from a conflict-free point of view, since:
\[
\pi_i + \Delta_i \leq \pi_j \Rightarrow \lfloor \pi_i \rfloor + \lceil \Delta_i \rceil \leq \lfloor \pi_j \rfloor 
\quad (15)
\]

**Fig. 9 - The dive-and-fix heuristic for the MinWL problem**

The rationale behind using such an approach (instead of, e.g., pushing the dive-and-fix heuristic to its limit, including also the setting of \( \pi, \Delta \) into that) is twofold: on one hand, it is considerably faster, since it requires solving a single LP to fix as many as \( 2E \) variables. On the other hand, for many instances the solution obtained from the linear relaxation was already integer, meaning that the reduced integer MinWL problem has a strong linear relaxation.

The two blocks can be inserted to replace the MinWL solver in the iterative solution scheme of Fig. 5, i.e. at each iteration a new dive-and-fix and reduced MinWL are solved. As we will show later on in Section 7, this alone would guarantee much smaller solution times for the E2EFP problem, at the price of a tolerable degradation in the quality of the solutions. However, we can push efficiency further by a simple observation. In Fig. 8 the dive-and-fix block represents the most complex task (due to the fact that an LP problem has to be solved several times to fix all the conflicts), which accounts for most of the time consumed by an iteration. Accordingly, when a link schedule is found infeasible from a delay point of view, instead of performing the whole sequence of dive-and-fix and reduced MinWL solving, we proceed as follows:

- we maintain an iteration counter \( i \), which is increased at every iteration
- if \( i \) is a multiple of a configurable parameter \( H \), both the dive-and-fix and the reduced MinWL are executed on the new problem instance as modified by the feedback;
- otherwise, only the reduced MinWL problem is solved, i.e. the previous conflict orientations are retained through the iterations.

The configurable parameter \( H \) is called conflict orientation frequency (COF).

Clearly, by increasing \( H \), the same number of iterations completes in a smaller time. However, as \( H \) increases, the quality of the solution degrades as well: in the limit case \( H = +\infty \), in fact, the conflict orientations would be assigned once and for all (without taking into account any further feedback), which would severely constrain (hence hamper) the search for a feasible schedule. However, as we show in the next section, by tuning \( H \) a remarkable reduction of the computation time can be harvested with a negligible degradation in the quality of the solutions, which is ultimately what allows us to use this framework for online admission control.

### 7. Performance evaluation

In this section we evaluate the performance of the iterative algorithm, with its core (i.e., the MinWL problem) solved both optimally and suboptimally through the heuristic described in Section 6. The purpose of this section is twofold. First, we analyze the sensitivity of the iterative algorithm to the two knobs that a network engineer can tune, namely the feedback granularity \( K \) (for both an optimal and a heuristic solution of the MinWL), and the COF \( H \) (for the heuristic solution of the MinWL), so as to provide guidelines for setting the latter in practical cases. Second, we show that using an approximated solution for the MinWL problem allows one to solve the link scheduling problem within few hundreds of milliseconds, even in quite large networks (50+ nodes), and that this comes with a negligible loss in solution accuracy. In other words, there are few cases when optimally solving the MinWL produces a feasible schedule (whatever the time it takes) and the heuristic solution does not. Third, we exploit the solution algorithm to explore how the network and flow parameters affect schedulability.

The test set is a balanced binary tree of 31 nodes. 30 identical flows originate at nodes 0-29 and traverse the net-
work towards node 30. The capacity of each link is set to 9600.

CPLEX [22] was used as a solver for both the MinWL MILP problem and the LP problems derived from its relaxation in the heuristic approximation. We are aware that, in both cases, the computation times that we obtained may be biased by the interaction between the CPLEX solver and our C++ code (which involves preparing and passing large data structures), and that they could probably be reduced by optimizing the entire code structure. However, this is unlikely to change their orders of magnitude, which are the relevant figures for judging the practicability of our solution schemes in an online admission control context.

As a first study, we show that the feedback granularity $K$ has an impact on the overall computation time: the higher $K$ is, the smaller the number of iterations before a positive/negative answer to the E2EFP is found. This is shown in Fig. 10, where the execution time for the same instance is plotted as a function of $K$. The instance is a network with 30 homogeneous flows with $\sigma = 150$, $\rho = 300$ and $\delta = 22$, for which the E2EFP is indeed feasible. The first noteworthy point is that, whatever the value of $K$, the solution time is in the order of some seconds, i.e. infeasible for online admission control. Furthermore, we observe a non monotonic dependency on $K$. This is due to the fact that we increase the rates in discrete steps. However, as confirmed by the logarithmic interpolation (dashed line), the trend is generally decreasing: a larger $K$ entails a smaller number of iterations, hence a smaller solution time. However, as $K$ grows larger, some instances are declared infeasible: the dotted red line represents values of $K$ for which the instance is not solved. This is because too gross a feedback prevents the iterative algorithm from exploring the solution space effectively. In this case the time reported is the one required to declare the E2EFP as infeasible.

Fig. 10 – Solving time as a function of $K$, (MinWL optimally solved)

We now move to considering the effectiveness of the heuristic solution of the MinWL. We want to show what we pay for a faster solution time, amenable to online admission control. Preliminarily, we show that the heuristic approximates the optimal MinWL solution acceptably well. Fig. 11 shows a histogram reporting the number of solved instances against the relative gap with respect to the optimal MinWL solution value. 100 instances were created by generating the flow rate from a uniform distribution within $[50,300]$. The figure shows that the gap is within 20%, which we consider acceptable for our purposes. Note that 97 instances out of 100 were solved.

Fig. 11 – Percentage of solved MinWL problems as a function of the relative gap between the heuristic and the optimal solutions

Then, as the MinWL is not solved alone, but instead set into an iterative framework, we show how solving it suboptimally changes the solution effectiveness and time for the E2EFP. Fig. 12 shows the percentage of E2EFP instances which can be solved using the heuristic approach, for different values of $H$, for $K = 100$ (above) and $K = 500$ (below).

Fig. 12 - Percentage of solved instances with the heuristic approach for $K=100$ (above) and $K=500$ (below)

The number of E2EFP instances is equal to 100; for each one, rates and bursts of the flows are generated uniformly between $[10,300]$ and $[100,1000]$ respectively, with $\delta$ ranging from 20 to 60. For the same instances Fig. 13 reports box plots showing the distribution of the solution times for six values of $H$.

Fig. 13 – Distribution of the solution times with the heuristic approach for $K=100$ (above) and $K=500$ (below)

The results clearly show that, as $H$ grows larger, the number of instances which can actually be solved decreases. This is due to the fact that the conflict orientations are considered less often, reducing the space for optimization due to the increased feedbacks. However, the computation times decrease as well. We can easily tune $H$, e.g. around 5, so that the number of solved instances is still high, whereas the solving time is below an acceptable threshold (i.e. 100-200 ms) most of the times.

We have also tested mesh networks of larger scales. We have generated regular tree topologies with an increasing number of nodes: binary trees with 3 to 6 levels (i.e., 7, 15, 31, 63 nodes) and a 6-level binary tree with an incomplete bottom level (47 nodes). For each topology, 100 instances were generated with parameters taken uniformly within the values specified in Table 3. These intervals were determined so as to allow the solver to cycle several tens of times in all topologies, which makes a time comparison more well founded.

Fig. 14 shows the average (line) and 95th percentile of the solving time (markers above the line), using $K = 100$, $H = 10$. As the figure clearly shows, we have solving times within a few hundred milliseconds up to more than 50 nodes. These can be considered within an acceptable range for online admission control, also taking into account that smaller times might be reached by just optimizing the solver.
As a last contribution, we exploit our iterative solution scheme to provide some insight into the dependencies among the various parameters. We tested the iterative algorithm changing both the burstiness and the sustainable rate for all the flows. Fig. 15 shows the maximum achievable sustainable rates $\rho_i$ against a requested end-to-end delay bound, with different values of burstiness, in the following scenarios:

a) Balanced binary tree of 31 nodes

b) Symmetric tree of 31 nodes, where the number of children nodes at each level is respectively 3, 3, 2.

c) Random (unbalanced) tree of 30 nodes

The MinWL problem is optimally solved in all the three cases.

The graphs exhibit a threshold behavior: for each burst value, there exists a minimum end-to-end delay bound that can be guaranteed. Once this threshold is approached, the maximum achievable rate increases rather steeply, quickly reaching the point where the links closer to the gateway are saturated. In general, the saturation point will depend on both the capacity of those links and their degree of interference, i.e. their number of neighbors in the conflict graph, and – obviously – on the traffic. For symmetric topologies (i.e., cases a) and b)), a quick back-of-the-envelope computation allows one to compute an upper bound on the maximum achievable rate. For instance, in the binary tree case, the two links close to the gateway are mutually conflicting, hence their maximum available rate is $9600/2=4800$. Each of those links carries the flow from 15 homogeneous sources, hence the maximum achievable rate should be $4800/15=320$. The alert reader can easily check that the same value is also the theoretical maximum for topology b). As Fig. 15 shows, the maximum schedulable rate using our approach, achieved for large enough deadlines depending on the bursts, is in both cases very close to the theoretical maximum (negligible differences being in fact accountable to the tolerance value set in the solver). In all the three cases, the threshold delay bound is monotonically increasing with the burst value.

In Fig. 16, the minimum schedulable delay bound is plotted against the burst of the flows. We set $\rho = 300$ for topologies a) and b) and $\rho = 200$ for the random tree c). Bursts are the same for all flows. The figure shows that the minimum deadline increases with the burst, which is expectable. Moreover, in case b), the deadline are considerably smaller, possibly due to the fact that the tree is shallower.

The above analysis shows that the relationship between flow burst and the requested delay bound plays a major role on the feasibility of the E2EFP problem. On the other hand, the sustainable rates are less important. This is partly due to our design choice of devising a scheduling that minimizes the latencies of network links: recalling the delay bound in (3), it is clear that minimizing the latencies, i.e. the scheduling delay components, the flow burstiness, which directly influences the queuing delay component, plays a predominant role on the achievable delay bounds.

As a final note, we remark that a comparative test of the effectiveness of our iterative algorithm is impossible, since it is not possible to solve the E2EFP optimally, even for trivial instances (4-5 nodes). As already pointed out in the previous sections, the structure of the E2EFP is such that general purpose solvers are not guaranteed to compute solutions in reasonable times even for such instances, let alone for larger scales. This is due to the non differentiable mixed integer-linear nature of the problem, and cannot be avoided.

8. Related Work

In this section we review the available related work on link scheduling in WMNs. Most of works on link scheduling in WMN take into account rate requirements, rather than end-to-end delay bounds. This is the case, for instance, of [29]-[33]. When a flow is provided with a minimum guaranteed rate (which is no smaller than its average sending rate), its worst-case delay is for sure finite. However, there is no guarantee that the latter will be below a pre-defined bound. The link scheduling algorithms described above reserve a rate equal to the flow’s $\rho_i$. It may instead be necessary to reserve a considerably larger rate if tight delay requirements are specified. Some recent works, however, include TDMA delay into link scheduling computations, either to minimize it (e.g. [36], [17]) or to guarantee a maximum TDMA delay...
TDMA delay is the sum of the waiting times at every hop, i.e. the time it takes for a packet to travel from the source to the destination, assuming that it is never queued behind other packets. As queuing is a component (and often the dominant one) of the end-to-end delay, especially with Variable Bit Rate (VBR) traffic, there is no guarantee that such algorithms can actually find a schedule that meets all the deadlines if there exists one. To better show the point, we compare our schedules with the optimal ones derived from [17]. In that work, the activation of each link is computed based on the rate of the flows traversing it, and activations are sequentialized so as to minimize the maximum TDMA delay. Consider a 15-node binary tree, with homogeneous traffic originating at each node. Fix $\delta = 35$, $\rho = 300$, and let the burst of the flows vary as $4000 \leq \sigma \leq 5000$. We plot the value for $V_{\text{max}}$ obtained by: i) solving the MinWL optimally, with $K=100$, ii) using the heuristic solution with $H=5$, and iii) using the optimal solutions given by [17] in the same settings. As Fig. 17 shows, according to [17] the above traffic cannot be guaranteed the requested deadline. On the other hand, there exists a way to schedule that traffic, and both the optimal and the heuristic frameworks find it in few iterations (the number of iterations is reported in Table 4). This is because [17] optimizes only conflict orientations ($o_{ij}$) and activation instants ($e_{\pi}$), neglecting the activation durations, which instead play a key role.

Fig. 17 – Comparison between the optimal solutions to a TDMA-delay minimization problem and the solutions obtained with our framework

Table 4 – Number of iterations required for the convergence

Other works provide frameworks for computing delay bounds a posteriori, after link scheduling has been planned. In [37] authors define the odd/even link activation and routing framework, and employ internal scheduling policies at each link so that the end-to-end delay bound along a path is roughly double the one obtained in a wired network of the same topology. Authors of [38] show that using throughput-optimal link scheduling and Coordinated-EDF to schedule packets within each link, rate-proportional delay bounds with small additive constants are achieved. Our goal is instead to have pre-specified, arbitrary delay bounds respected through link scheduling.

9. Conclusions and future work

This paper has addressed the problem of link scheduling in Wireless Mesh Networks. Unlike previous work, this one has brought end-to-end delay bounds in the picture, i.e. attempted to define a link scheduling algorithm able to guarantee pre-specified delay bounds. To our knowledge this is the first attempt to formulate such a problem. We have shown that the above problem is hard to solve, being mixed integer/non linear and with non differentiable delay bound constraints. We therefore adopted a heuristic iterative solution scheme, based on: i) a mixed integer-linear formulation of the link scheduling problem, and ii) a feedback module which tests whether the delay bound constraints are met in the current schedule. Depending on how the mixed integer-linear problem is solved (whether optimally or suboptimally, using dive-and-fix and linear relaxation), the overall solution computation times change by orders of magnitude. More specifically, the suboptimal approximated solution allows link schedules to be computed in hundreds of milliseconds in large-scale mesh networks (i.e., up to 60 nodes), without losing much as far as solution quality is concerned with respect to the optimal approach.

Future work on the same topic will include bringing back routing in the picture, i.e., extending the range of topologies that can be analyzed beyond the sink tree. This would imply solving a joint routing–scheduling problem with end-to-end delay constraints.

References


works”, in Proc. WMuNeP’05, Montreal, CA, Oct. 13, 2005


[38] P. Jayachandran, M. Andrews, “Minimizing end-to-end Delay in Wireless Networks Using a Coordinated EDF Schedule”, INFOCOM 2010

10. Appendix

Table 5 – Symbols used within the paper