Dynamic QoS-Aware Traffic Planning for Time-Triggered Flows with Conflict Graphs

Technical Report TR-2021-01

Jonathan Falk, Heiko Geppert, Frank Dürr, Sukanya Bhowmik, and Kurt Rothermel

University of Stuttgart
Institute for Parallel and Distributed Systems (IPVS)
Universitätsstraße 38
70569 Stuttgart, Germany

May 5, 2021

Many networked applications, e.g., in the domain of cyber-physical systems, require strict service guarantees, usually in the form of jitter and latency bounds, for time-triggered traffic flows. It is a notoriously hard problem to compute a network-wide traffic plan that satisfies these requirements, and dynamic changes in the flow set add even more challenges. Existing traffic-planning methods are ill-suited for dynamic scenarios because they either suffer from high computational cost, can result in low network utilization, or provide no explicit guarantees when transitioning to a new traffic plan that incorporates new flows.

Therefore, we present a novel approach for dynamic traffic planning of time-triggered flows. Our conflict-graph based modeling of the traffic planning problem allows to reconfigure active flows to increase the network utilization, while also providing per-flow QoS guarantees during the transition to the new traffic plan. Additionally, we introduce a novel heuristic for computing the new traffic plans. Evaluations of our prototypical implementation show that we can efficiently compute new traffic plans in scenarios with hundreds of active flows for a wide range of scenarios.

1 Introduction

Driven by so-called cyber-physical applications in domains such as the Industrial Internet of Things (IIoT), automotive systems, or smart grid infrastructures, we observe a strong trend towards deterministic real-time communication. This trend manifests itself in recent initiatives from major standardization bodies: For LANs, the IEEE has standardized several extensions to IEEE Std. 802.3 networks (Ethernet), subsumed under the term Time-Sensitive Networking (TSN). TSN in particular describes extensions such as the Time-Aware Shaper (TAS) to support time-triggered, i.e., scheduled traffic flows [1]. A similar quest for deterministic networking can be observed for routed networks as standardized by the IETF Working Group for Deterministic Networking (DetNet) [2].

Motivated by these trends, we address the problem of traffic planning, i.e., how to compute routes and schedules that provide these deterministic guarantees. These schedules could, e.g., be implemented by reserved transmission windows using the TAS. It turns out that already the calculation of transmission schedules alone often is an NP-hard problem, for instance, related to the well-known Job Shop Scheduling Problem [3], and if the routes are not given, we have to add a dimension (space) to the scheduling problem. This traffic planning problem is often addressed for static, a-priori known sets of flows. However, dynamic changes in the flow set often arise naturally, e.g., when attaching or physically moving a sensor or machine in a smart factory [4], or—in the future—in the backbone of vehicular communication scenarios [5, 6].

If we consider dynamic scenarios where flows can be added dynamically, the existing approaches come with serious drawbacks. Early approaches for static traffic planning using Integer Linear Programming (ILP) or Satisfiability Modulo Theory (SMT) [7, 8], although theoretically finding exact solutions or a proof of infeasibility, practically are often limited to small scenarios, and exhibit long runtimes. Therefore, fast(er) heuristics [3, 9] for static planning have been presented which raises the question whether static planning approaches could be employed in the context of dynamic traffic planning. Conceptually, there are two straight-forward options:

We could "freeze" every active flow, and run the static traffic planning only for the new flows. This amounts to defensive planning, and we can find defensive planning approaches in the literature [10], too. Defensive planning approaches do not change the configuration of active flows in order to add new flows. In other words, defensive planning takes the configuration of active flows for granted and uses the remaining network resources for routing and scheduling new flows. While defensive planning never affects the quality of service (QoS) of active flows, it might utilize network resources in a suboptimal way.

In contrast, offensive planning allows for reconfigurations of some active flows when adding new flows. While those approaches guarantee deterministic communication in the long run, they may cause short-term degradation of QoS, where both the degree and time of degradation is limited. While offensive planning allows for better utilization of network resources, it requires applications that can tolerate controlled fluctuations of QoS. A straight-forward way to perform offensive planning with static planning approaches is executing the static traffic planning again from scratch each time new flows are to be added. However, this provides no control over QoS degradation experienced by an individual flow. Worse yet, we might even have to suspend the emission of new packets for some time to ensure that the new traffic plan is not disturbed by old packets or, if technically possible, drop these old packets, and usually we cannot guarantee that the new plan still includes all priorly active flows.

To overcome these issues, we propose a different approach for dynamic traffic planning that allows reconfiguring active flows (offensive planning) while also providing bounded QoS degradation during the transition to the new traffic plan. In detail, the contributions of this paper are twofold:

- We present a novel approach for dynamic traffic planning for time-triggered flows using the zero-queuing principle [3, 11]. Our approach supports offensive traffic planning with seamless transitions from the old traffic plan to the new traffic plan, i.e., a traffic plan update does not require artificial pauses of sender nodes or dropping packets. Additionally, our approach can guarantee per-flow bounds on the QoS degradation during the transition from the old traffic plan to the new traffic plan.

- We present a novel heuristic for the underlying optimization problem which we have to solve to obtain a new traffic plan. In particular, we use a conflict-graph based modeling [12] for the traffic planning. The conflict-graph based modeling is well-suited for dynamic traffic planning because the conflict graph embeds a large portion of the knowledge about the solution space from the previous traffic plan, and thus reduces the effort required to compute the new traffic plan. Computing a new traffic plan requires to solve a variant of the independent colorful vertex set problem with weighted colors. Our evaluations show that our novel heuristic can solve this problem for hundreds of flows in a fraction of the time required by a state-of-the-art optimizer.

Following the related work discussion in Sec. 2, we introduce our system model, modeling concepts, and problem statement in Sec. 3-4. We describe the traffic planning approach in Sec. 5–8, evaluate our approach in Sec. 9 and conclude the paper in Sec. 10.

2 Related Work

Here, we concentrate on related work for traffic planning for deterministic networking.

Existing work on *static* planning can be roughly sorted in two categories: 1) approaches that rely at the core on generic constraint programming frameworks at such as Integer Linear Programming (ILP) [13, 14, 15, 7, 16] or Satisfiability Modulo Theories (SMT) [17, 18, 19, 20, 8] and 2) approaches that use custom heuristics [3, 9, 21], or a combination of both [12]. Static planning approaches require adaptions to be used for dynamic traffic planning. Modifications towards defensive planning operate analog to incremental approaches [10, 22] and ensure that "nothing bad" happens when new flows are added, because the traffic plan from each previous step remains immutable for the future. However, defensive traffic planning prohibits to "correct" any past decision that may turn out to have been sub-optimal when facing new flows. This can result in low network utilization if a flow that has been added early on obstructs new flows.

This could be overcome by offensive planning that allows to reconfigure active flows. In theory, one can implement offensive planning by executing a "fast" static planning algorithm each time new flows are added. The first problem is, how to ensure that this does not oust currently active flows, because—without additional precautions—each update may remove active flows from the network, e.g., to maximize the total amount of flows. Even if that was tolerable, we still have to put severe restrictions on the network topologies and flow-deadlines, or, in general, we need a "guard interval" between two subsequent traffic plans to protect the new packet from interfering with old packets, if old packets cannot or must not be dropped in the network. By nature, static planning algorithms do not account for old packets from a previous traffic plan which results in the mentioned problems in dynamic scenarios.

Our novel approach fills the gap between defensive planning approaches and the lack of offensive planning approaches which require no guard interval or packet drops. Additionally, our approach can incorporate application-specific per-flow bounds on the (temporary) QoS degradation during an update of the traffic plan. On top of that, our modeling approach allows reusing most of the state which encodes the solution space from the previous step.

3 System Model

We consider traffic flows in packet-switched data networks. A traffic flow is a directed sequence of packets that is transmitted from a source node to a destination node through the network. The network consists of infrastructure nodes (switches, routers), source nodes and destination nodes, and controller nodes. Infrastructure nodes forward data packets via point-to-point links which connect nodes. Source node and destination node are the origin and target of the packets of a flow, respectively. Packets are sent and forwarded according to the global *traffic plan*. The traffic plan is computed and disseminated by the planner, which is situated at the logically centralized controller and has global view onto the network.

3.1 Flow Dynamics

Whenever a set of new flows are to be added to the network, the planner computes a new traffic plan. Therefore, over time the planner computes a sequence of traffic plans, where a newly computed traffic plan becomes the current plan, which replaces the old one. A traffic plan, say p, defines for the set of flows admitted to the network when and along which routes the packets of the active flows are transmitted.

We will use $\text{ActiveF}(p)$ to denote the set of active, i.e., admitted flows of plan p. In the following, we concentrate on the case that the planner only attempts to add new flows to the network. Removing active flows is trivial from the perspective of the planner, cf. Sec. 6.2. Let $\text{ReqF}(p)$ be the set of flows the applications request to be added to the network, i.e., $\text{ReqF}(p)$ are the requests received by the planner while p is valid. By processing this request, the planner generates a new traffic plan p' with $\text{ActiveF}(p') \setminus \text{ActiveF}(p) \subseteq \text{ReqF}(p)$, where p is the plan preceding p'. In the following, we denote the current traffic plan by p (which becomes the old traffic plan after the update), and the new traffic plan by p'.

3.2 Time-Triggered Traffic Flow

The notion of a traffic flow represents a directed communication channel from one source node to one destination node. In the following, we focus on time-triggered flows that carry time-sensitive, important information requiring deterministic QoS and no packet drops under normal operation conditions, i.e., in absence of external failures such as power-loss, physical disturbances, which are out of scope. Starting from the time of its activation, the source node of an active flow emits one packet per transmission cycle. A transmission cycle is an interval of length t_{cycle}. Transmission cycles start at $k \cdot t_{\text{cycle}} + t_0, k \in \mathbb{N}$ where t_0 is a common time-reference. Source, destination, packet-size, and t_{cycle}, and end-to-end delay are flow *parameters* and do not change during a flow's lifetime.

When an application requests a flow, it specifies the parameters in the request for the planner. Conversely, the planner assigns a *route*, which connects source node and destination node, and the *phase* ϕ to each active flow when computing the traffic plan. The value of ϕ is used by the source node to schedules its packet transmissions at $k \cdot t_{\text{cycle}} + \phi + t_0, k \in \mathbb{N}$. The valid range of ϕ is restricted to $[0, t_{\text{cycle}} - t_{\text{trans}}]$ with $t_{\text{trans}} = \frac{\text{packet size}}{\text{bandwidth}}$ being the transmission delay. Thereby, the transmission of a packet on the outgoing link at a source nodes is confined within the transmission cycle boundaries. Once the source node has sent a packet, the packet is forwarded by infrastructure nodes to the destination node. Unless ambiguous, we omit the flow indices of flow properties, i.e., we use t_{cycle} instead of $t_{\text{cycle}}(f)$.

3.2.1 Flow Configuration

A flow configuration is one assignment of values to the phase and route of a flow. Since a continuous time variable ϕ yields infinitely many flow configurations, we make some practical assumptions: As proposed in [12], we use discrete time with appropriately mapped granularity, e.g., with 1 µs resolution. Secondly, we also restrict the routing options to a set of candidate paths. Each candidate path must not only be loop-free, but is also restricted in length by the end-to-end delay constraint for each flow. We explain in Sec. 3.4 that zero-queuing provides a simple relation between path length and end-to-end delay t_{e2e}. Depending on the scenario, e.g., highly meshed network and "large" end-to-end delay bounds, we might still end up with impractical many candidate paths. Therefore, we also upper-bound the number of candidate paths. Upon receiving a request to add a flow, the planner then once computes a set of candidate paths for that flow, and assigns a path identifier π to each candidate path of the flow. In our case, the planner uses Yen's k-shortest path algorithm [23] to compute the candidate paths. Hence, we can define a flow configuration with a tuple of integers from a finite set.

Definition 1 (Flow Configuration)**.** *The tuple* (f, ϕ, π) *represents a flow configuration.*

Remember, offensive planning allows reconfiguring active flows while a new plan is established. Consequently, a flow may have different configurations over time. The notion $\text{config}(f, p)$ identifies the configuration of flow f while p is valid. If the planner decides to reconfigure f in a succeeding plan p', then $\text{config}(f, p) \neq \text{config}(f, p')$. We say that each reconfiguration of a flow generates a new version of this flow, where each version exists as long as the corresponding plan is valid. To be able to distinguish the various flow versions, we will use the notion $\langle f, p \rangle$, where flow f is associated with plan p, i.e., $\text{config}(f, p)$ is the configuration of $\langle f, p \rangle$.

3.3 Node Capabilities

We consider infrastructure nodes with store-and-forward behavior and FIFO queuing semantics per output port. Infrastructure nodes provide facilities to forward packets along *explicitly* defined routes, e.g., via SDN flow tables, VLAN tagging, explicit path control in IEEE Ethernet networks [24], etc. Source nodes support time-triggered packet scheduling, i.e., it is possible to explicitly specify the time of a packet transmission. Such functionality can be implemented in hardware, e.g., the "LaunchTime"-feature in commodity NICs, or in software, e.g., TAPRIO or ETF queuing discipline in Linux. Consequently, all nodes need synchronized clocks, e.g., using protocols such as Precision Time Protocol (PTP). Support for multiple traffic classes, and time-triggered packet scheduling (e.g. TAS, TTEthernet, etc.) in the infrastructure nodes is beneficial and can be used to protect the scheduled transmission windows from cross-traffic due to unscheduled flows. (Note that schedules and reserved transmission windows for mechanisms such as the TAS can effectively be derived from the phase ϕ of the corresponding configuration [12]).

Before a node sends or forwards packets of a particular flow f, the controller propagates the necessary flow information corresponding to $\text{config}(f, p)$ to the respective nodes in the network. What this entails is technology-specific, but includes routing entries and packet transmission schedules. The flow information in the nodes, in particular routes and schedules, can be updated at runtime by the controller. Flow information updates shall not affect in-flight packets to prevent network update issues such as black-holing. This means, infrastructure nodes temporarily may have multiple sets of flow information for each version of a reconfigured flow and deliver each packet according to the flow configuration that was used by the source node when sending the packet. This can be

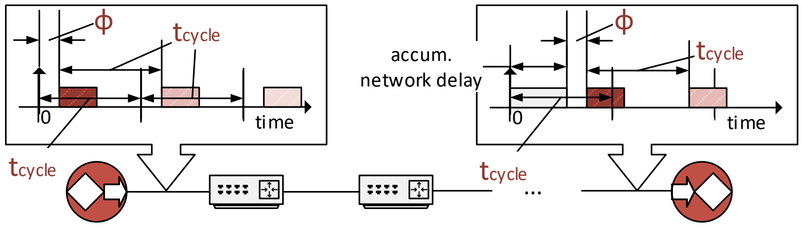


Figure 1: Zero-queuing: Packets accumulate deterministic delay on the route to the destination.

achieved, e.g., by tagging packets with additional metadata, address schemes, etc. The controller lazily purges obsolete flow information sets from the nodes once all old packets have been received by the destination nodes.

We consider a network where all links operate with the same transmission speed. We denote the processing delay of an infrastructure node by t_{proc}, and the propagation delay on a link by t_{prop}. These values relate to hardware properties (switching fabric, cable length, etc.) and are therefore considered constants. While it is not necessary for our approach that all nodes and links induce the same processing delay and propagation delay, respectively, we omit node and link identifiers for t_{proc} and t_{prop} in the following, i.e., assume they have same value network-wide.

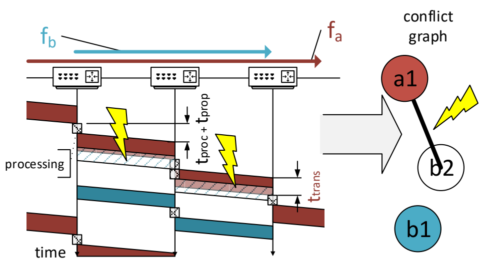
3.4 Zero-Queuing

Our approach uses the zero-queuing principle [3, 11] where packets must not be buffered in the network. Instead, packets always enter empty queues at each network element and are immediately forwarded to the next hop. Zero-queuing can be achieved by a global coordination (here: in the form of the traffic plan) of packet transmissions. If two flows violate the zero-queuing constraint, we say they *interfere*.

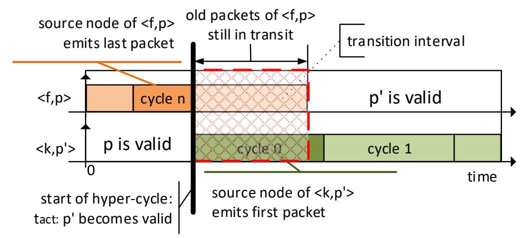
Fig. 1 depicts a packet sequence of a time-triggered flow at different points in the network with zero-queuing. The source node sends one packet per cycle on its outgoing link starting at $t_0 + \phi + k \cdot t_{cycle}, k \in \mathbb{N}$. At each hop, packets accumulate only the (inevitable) delay which consists of the time between receiving a packet and completing the forwarding of the packet on the next link. The per-hop delay $t_{\text{perhop}} = t_{\text{trans}} + t_{\text{prop}} + t_{\text{proc}}$ can be computed from t_{trans} and the network properties t_{prop} and t_{proc}. Thus, zero-queuing enables transmissions with bounded end-to-end delay $t_{\text{e2e}} = t_{\text{trans}} + t_{\text{prop}} + l \cdot t_{\text{perhop}} + t_{\text{src}} + t_{\text{dst}}$ that depends on the number of hops l in-between source and destination (i.e., l is the number of infrastructure nodes on the route), and processing delays of source and destination node (denoted by t_{src} and t_{dst}), respectively.

4 Concepts and Problem Statement

We use a conflict-graph based approach to compute the new traffic plan. Conflict-graph based modeling for static traffic planning was introduced in [12]. From a birds-eye view, the conflict-graph based modeling represents flow configurations and their relations by vertices and edges in a so-called conflict graph. A traffic plan can then be obtained from the conflict graph by searching for a set of conflict-free configurations.



(a) Configurations and conflict: Flows must traverse all links without buffering.



(b) Old packets from p may interfere with new packets from p' during the *transition interval*.

Figure 2: Flow interference can be caused by conflicting configurations and traffic plan updates.

Before we formally state our problem, we explain how these conflict-graph concepts relate to our planning problem and extend these concepts for dynamic scenarios.

4.1 Configuration Conflict

Intuitively, we have a conflict between two configurations if packets would have to be transmitted *simultaneously* on the same link to arrive at their respective destination without queuing. Since packet transmission are serialized at each output port, we define configuration conflicts as a violation of the zero-queuing constraint.

Definition 2 (Conflict)**.** *Let c_1 and c_2 be two different configurations. The two configurations are in conflict, if any packet sent with c_1 would be buffered due to a packet sent with c_2, or vice versa.*

This is illustrated in Fig. 2a. There is a conflict between configuration $a1$ for flow f_a and configuration b_2 for flow f_b, since the transmission at the source of f_b is scheduled too early, and packets sent with b_2 would have to be buffered until the transmission of packets from f_a is finished—which violates the zero-queuing principle. In contrast, configuration $a1$ und $b1$ are conflict-free due to the increased phase for packets of f_b.

The zero-queueing constraint allows us to conflict-check pairs of configurations independently in parallel. To compute whether two configurations conflict we have to check if the scheduled packet transmissions are always temporally isolated on common edges. Due to the cyclic property of the packet transmissions, we only check an interval of the length of a hyper-cycle of the two flows, i.e., the least common multiple of their t_{cycle}-values. However, packets may take longer than the hyper-cycle to reach the destination. Therefore, we use modulo arithmetics to “fold” transmissions crossing the hyper-cycle bounds back to the start of the interval.

4.2 Conflict Graph

We encode the relations and constraints between the configurations in a conflict graph. A flow configuration is represented by a vertex in the conflict graph, and conflicts between configurations of *different* flows are represented by edges in the conflict graph. For example, in Fig. 2a there is an edge between configuration *a1* and configuration *b2*.

A new version of the conflict graph is generated by modifying the old one whenever a new traffic plan is to be computed. As will be seen below, a new version of the conflict-graph is generated by adding/removing configurations to the previous version and/or locking/unlocking configurations. In the following, $\mathbf{G}(p)$ identifies the version of the conflict graph that is associated with plan p, i.e., $\mathbf{G}(p)$ has been used to compute p.

The conflict graph includes for each flow a set of potential configurations, which we will call the flow's candidate configurations, or candidates for short. We will use $\mathrm{cand}(f, p)$ to denote the candidate set of flow f in $\mathbf{G}(p)$. We can interpret $\mathbf{G}(p)$ as a colored graph.

Definition 3 (Conflict Graph)**.** *The conflict graph $\mathbf{G}(p)$ is an undirected, vertex-colored graph where all configuration vertices in cand(f, p) are colored by f. Two configuration vertices in $\mathbf{G}(p)$ are connected by an undirected edge if and only if they belong to different flows and there is a conflict between the two corresponding configurations.*

To insert a configuration to $\mathbf{G}(p)$, we add it to the vertex set of $\mathbf{G}(p)$ and add an edge to every other configuration in $\mathbf{G}(p)$ which conflicts with it. To remove a single configuration from $\mathbf{G}(p)$, we remove the configuration from the vertex set of $\mathbf{G}(p)$, and delete all edges that exist to the remaining configurations in $\mathbf{G}(p)$.

4.3 Global Traffic Plan

From the perspective of the planning problem, a traffic plan p provides a phase and route for each flow f in $\mathrm{ActiveF}(p)$ such that all packet transmissions are isolated temporally and/or spatially, and thus satisfy the zero-queuing principle. In other words:

Definition 4 (Traffic Plan)**.** *A traffic plan p is a set of conflict-free configurations for ActiveF(p) in $\mathbf{G}(p)$ which contains exactly one configuration for each flow in ActiveF(p).*

We say that traffic plan p admits flow f if p contains a configuration for f, i.e.,

$$\mathrm{admits}(f, p) = \begin{cases} 1 : \text{if } \exists \text{ configuration for } f \in p \\ 0 : \text{if } \nexists \text{ configuration for } f \in p \end{cases} . \tag{1}$$

Definition 5 (Independent Set)**.** *Denote by $\mathcal{V}$ the vertices in $\mathbf{G}(p)$, and denote by $\mathcal{E}$ the edges. $\mathcal{C} \subseteq \mathcal{V}$ is an independent subset of vertices in the conflict graph $\mathbf{G}$ iff $\forall u, v \in \mathcal{C} : (u, v) \notin \mathcal{E}$.*

With the correspondence vertex $\leftrightarrow$ configuration and edge $\leftrightarrow$ conflict, it is easy to see that a traffic plan corresponds to a set of independent vertices in the conflict graph [12].

Note that $\mathbf{G}(p)$ possibly contains many different sets of conflict-free configurations. While it is intuitively clear that we prefer to find a set of conflict-free configurations that admits all flows, this may not always be possible. Therefore, we specify the objective that guides the search for the configuration in Sec. 4.5.

4.4 Transition Interference

Assume that we have a new traffic plan p' and want to replace the traffic plan p with p'. We call the time a new traffic plan p' becomes valid its activation time t_{act}. To ensure that no source node is in the middle of a packet transmission at the activation time of p', and since the flows in $\mathrm{Active}(p)$ may have different cycle times, we choose the start of a fresh hyper-cycle of all active flows in $\mathrm{ActiveF}(p)$ as activation time, i.e., $t_{\mathrm{act}} = t_0 + k \cdot \mathrm{lcm}_f(t_{\mathrm{cycle}}(f))$.

However, at the time when the new traffic plan p' becomes valid, the remaining yet-to-be-delivered packets from $\langle f, p\rangle$, $f \in \text{ActiveF}(p)$ are still in the network. If we do not take this into consideration, this can result in *transition interference*, i.e., flow interference between the remaining old packets from $\langle f, p\rangle$, $f \in \text{ActiveF}(p)$ and the first packets from $\langle k, p'\rangle$, $k \in \text{ActiveF}(p')$.

Definition 6 (Transition Interference)**.** *Let p denote the traffic plan valid before t_{act} and let p' denote the new traffic plan with activation time t_{act}. Transition interference is defined to be between $\langle f, p\rangle$, $f \in ActiveF(p)$ and $\langle k, p'\rangle$, $k \in ActiveF(p')$ if any packet sent by the source node of $\langle f, p\rangle$ prior to t_{act} is buffered due to a packet sent by the source node of $\langle k, p'\rangle$ from t_{act} on, or vice versa.*

We can algorithmically check whether transition interference occurs between $\langle f, p\rangle$ and $\langle k, p'\rangle$: If the route of $\langle f, p\rangle$ and the route of $\langle k, p'\rangle$ have common links, and any packet from $\langle f, p\rangle$ sent before t_{act} is scheduled for transmission at any point in time when any packet sent by $\langle k, p'\rangle$ from t_{act} on is scheduled on a common link, transition interference occurs.

Transition interference is restricted to a possible zero-length time interval, the *transition interval*, which starts at t_{act} and is upper-bounded by the end-to-end delay of $\langle f, p\rangle$, cf. Fig 2b. In the following, we only consider the case where transition intervals of different traffic plan updates are mutually exclusive, i.e., we only consider a traffic plan p and its immediate successor p'.

4.5 Problem Statement

Let p be the traffic plan for the flows in $\text{ActiveF}(p)$ computed from $\mathbf{G}(p)$, i.e., the packets of flows in $\text{ActiveF}(p)$ are transmitted according to p. Now applications request that a set of flows $\text{ReqF}(p)$ is added to the network. Our goal is to compute a new traffic plan p'.

To compute p', we have to construct the new conflict-graph $\mathbf{G}(p')$ and search for a set of conflict-free configurations (i.e., independent vertices). In our case, we want to ensure that once the planner added a flow to $\text{ActiveF}(p)$, this flow remains active until the application itself indicates the flow can be removed. This means, the planner shall not unsolicitedly evict any flow from $\text{ActiveF}(p)$. Consequently, the new traffic plan shall include a configuration for *every* flow in $\text{ActiveF}(p)$ and as many new flows as possible from $\text{ReqF}(p)$. The different importance of active flows and new flows results in the following objective.

Definition 7 (Traffic Plan Objective)**.** *Let $\mathbf{G}(p')$ be a conflict graph which contains candidates for all flows in $\text{ActiveF}(p) \cup \text{ReqF}(p)$ and $\text{ActiveF}(p) \cap \text{ReqF}(p) = \emptyset$. We want to find a new traffic plan $p' \subseteq \bigcup_f \text{cand}(f, p')$ that maximizes the objective*

$$\max_{p'} \sum_{f \in \text{ActiveF}(p)} \text{admits}(f, p') + \sum_{f \in \text{ReqF}(p)} \frac{1}{|\text{ActiveF}(p) \cup \text{ReqF}(p)|} \cdot \text{admits}(f, p'). \tag{2}$$

The factor $\frac{1}{|\text{ActiveF}(p) \cup \text{ReqF}(p)|}$ in Eq. 2 discounts the relative importance of flows in $\text{ReqF}(p)$. This means, any flow in $\text{ActiveF}(p)$ is more "valuable" than all flows $\text{ReqF}(p)$ combined. From a graph-theoretic perspective, this is a specific colorful independent vertex set problem where we want to find a set of independent vertices and each color has either unit weight or a weight inversely proportional to the total amount of colors.

5 Constructing the Conflict Graph

Next, we explain how the planner constructs the new conflict graph $\mathbf{G}(p')$ when processing $\text{ReqF}(p)$. With offensive planning, the planner has to consider both flow versions $\langle f, p\rangle$ and $\langle f, p'\rangle$ for each active flow f, where p' indicates the new plan replacing p. Remember that after installing p', packets from both $\langle f, p \rangle$ and $\langle f, p' \rangle$ may be in the network for some time. Therefore, it is not sufficient to only avoid conflicts between any new flow versions $\langle f, p' \rangle$ which would suffice in the static case. Instead, the planner additionally has to ensure that any $\langle f, p \rangle$ does not interfere with any other flow $\langle k, p' \rangle$. This leads to the following requirements:

R1 For all $k, f \in \text{ActiveF}(p')$, $\langle k, p' \rangle$ does not interfere with $\langle f, p' \rangle$ for $k \neq f$, i.e., active flows of the new plan do not interfere.

R2 For all $k \in \text{ActiveF}(p)$ and $f \in \text{ActiveF}(p') \setminus \text{ActiveF}(p)$, $\langle k, p \rangle$ does not interfere with $\langle f, p' \rangle$. That is, the active flows of the old plan do not interfere with the flows added to the new plan.

R3 For all $k, f \in \text{ActiveF}(p)$, $\langle k, p \rangle$ does not interfere with $\langle f, p' \rangle$. That is, the active flows of the old plan do not interfere with their versions in the new plan. Note, if $f = k$, then we consider the old and new version of the same flow, $\langle f, p \rangle$ and $\langle f, p' \rangle$. Since packets from $\langle k, p \rangle$ and $\langle f, p' \rangle$ may populate the network at the same time, we have to ensure that no transition interference occurs.

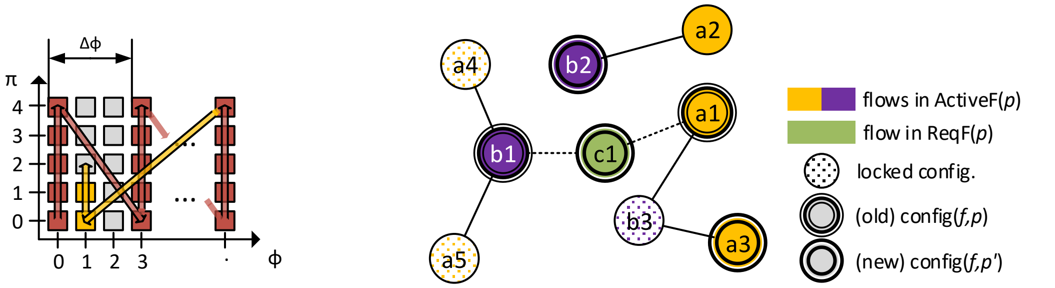
R1 is a basic requirement, which is sufficient for defensive planning. Allowing for reconfigurations of active flows, however, we must add R2 and R3 to ensure the zero-queuing constraint. In order to fulfill R1 the planner has to make sure that the configurations associated with flows in $\text{ActiveF}(p')$ are not in conflict. This can be achieved by *adding candidate configurations* for each flow in $\text{ReqF}(p)$. The expanded conflict graph is the basis for solving the planning problem as stated above.

Now let us see, how additionally R2 and R3 can be guaranteed. Note that the interference of two flows/flow versions implies that their configurations are conflicting, while the opposite is not true. For example, if we ensure that the packets of one flow or flow version only enter the network after all packets of another flow already left the network, these two flows do not interfere even if their configurations are conflicting. Therefore, R2 can simply be fulfilled by *postponing the transmission* of the added flows until all packets that belong to flow versions of p have died out. Assume the transmission of packets of each flow $\langle k, p \rangle$ with $k \in \text{ActiveF}(p)$ is stopped at time t. All packets sent by these flows have left the network by $t + \tau$, where $\tau \geq \max_f(t_{\text{e2e}}(f) - t_{\text{cycle}}(f))$ is greater or equal to the largest transition interval of any flow f. If transmission of each $\langle f, p' \rangle$ with $f \in \text{ActiveF}(p') \setminus \text{ActiveF}(p)$ starts not before $t + \tau$, then no flow $\langle f, p \rangle$ interferes with any flow added to p', even if their configurations conflict. Therefore, fulfilling R2 requires no special treatment when constructing the conflict graph.

Fulfilling R3 the same way as R2 is definitely not desirable. While postponing the start of added flows for a short time (usually a few µs) does not affect QoS, delaying data packets of an active flow might degrade QoS substantially. Therefore, we have to make sure that the planner cannot select configurations for active flows, which conflict with the old configurations of these flows. We achieve this by *locking those configurations* the planner must not select for the new plan. Note that the configurations to be locked depend on the old configuration of the active flows. Since the configuration of an active flow may change from plan to plan, configurations are unlocked after the new plan has been computed.

5.1 Adding Candidate Configurations

Since $\mathbf{G}(p)$ only contains candidates for flows in $\text{ActiveF}(p)$, the planner has to generate and inserts new candidate configurations for all flows in $\text{ReqF}(p)$ to the $\mathbf{G}(p')$. In principle, the planner



(a) New candidates are generated by traversing the ϕ-π-space.

(b) Example: 1-hop neighbors $a1, b1$ of active configs. are locked in $\mathbf{G}(p')$, $a2, a3, b2$ remain for the active flows.

Figure 3: The planner obtains new candidate configurations to add to the conflict graph by invoking the configuration generators (a) and locks configurations causing transition interference (b).

could add all candidates for each flow to $\mathbf{G}(p')$. However, we know from the static flow planning problem [12] that we can find a traffic plan for all flows even if the conflict graph contains just a subset of all possible candidates of each flow. We exploit this and add only a limited number (upper-bounded by a parameter n_{ub}) of candidate configurations for each flow in $\text{ReqF}(p)$ to the conflict graph.

Due to the nature of the traffic planning problem, it is often inherently difficult to identify promising candidate configurations a-priori. In this paper, we therefore use the following heuristic: For each flow, the planner has a stateful configuration generator that walks through the ϕ-π-space and returns a new configuration each time it is invoked. Starting at $\phi = 0$,$\pi = 0$, π is incremented until all configurations for a particular value of ϕ have been covered, before increasing ϕ by $\Delta\phi$. If $\phi + \Delta\phi$ exceeds the allowed range for ϕ, ϕ is reset to the next, lowest uncovered phase-value, cf. Fig 3a. In our case, the planner sets $\Delta\phi$ to the 75-th percentile of the transmission duration of all flows in $\mathbf{G}(p')$, hence we expect that one to two phase increments suffice to resolve a possible interference on a single link.

While it is obvious that we have to add candidates for flows in $\text{ReqF}(p)$, we can also increase the set of candidates for flows in $\text{ActiveF}(p)$ in $\mathbf{G}(p')$. Here, we add more configurations for each flow in $\text{ActiveF}(p)$ when processing $\text{ReqF}(p)$ until the configuration generator has traversed the ϕ-range of a flow for the first time. After that point, new configurations for active flows are only added if the planner could not admit all flows from the previous request, i.e., a simple feedback loop controls the growth of the conflict-graph. The larger $\text{cand}(f, p')$ the more of the solution space is covered, and the more alternatives for an active flow "to make way" for flow in $\text{ReqF}(p)$ exist. The planner could also add the additional configurations for flows in $\text{ActiveF}(p)$ when idle, i.e., while no $\text{ReqF}(p)$ has to be processed. Since the lifetime of active flows may differ, the planner should aim for balanced number of candidates per flow in the long-term.

5.2 Locking Configurations

$\mathbf{G}(p')$, generated as describe above, may include configurations that cause interference between the old and new versions of active flows, violating R3. We yet have to make sure that the planner cannot select configurations for active flows that violate R3. In detail, the following two conditions have to hold:

1. For each $f \in \text{ActiveF}(p)$, there exists no configuration $c \in \text{cand}(f, p')$ that results in a new version $\langle f, p' \rangle$ that interferes with $\langle f, p \rangle$, i.e., the planner cannot assign a new configuration to f, such that the old and new version of f interfere.

2. For each $f, k \in \text{ActiveF}(p)$, $k \neq f$ there exists no configuration $c \in \bigcup_k \text{cand}(k, p')$ that results in a new version $\langle k, p' \rangle$ which interferes with $\langle f, p \rangle$. That is, the planner cannot assign a new configuration to f, which results in interference with an old version of any other active flow k.

We achieve both conditions by locking all those configurations in $\mathbf{G}(p')$, which would violate one of the conditions if selected by the planner.

To fulfill the first condition, for each $f \in \text{ActiveF}(p)$ we have to visit each $c \in \text{cand}(f, p')$ in $\mathbf{G}(p')$ to check whether or not a new version of f that uses the visited candidate c results in transition interference with the old version that uses $\text{config}(f, p)$. If transition interference would occur, we have to lock the candidate. Obviously, this ensures that the planner cannot select a candidate c for $\text{config}(f, p')$ which could cause interference with $\langle f, p \rangle$. Note that the configuration graph includes only edges for conflicting configurations of different flows. Therefore, we maintain an additional data structure that allows efficient access to the candidates of a particular flow.

To fulfill the second condition, for each $f \in \text{ActiveF}(p)$ we have to visit all 1-hop neighbors of $\text{config}(f, p)$ in the conflict graph, i.e., all candidate configurations of any flow other than f that conflict with $\text{config}(f, p)$. If a visited neighbor c is configuration of another flow k in $\text{ActiveF}(p)$, the planner checks for transition interference between a new version of k with configuration c and $\langle f, p \rangle$. If there would be interference, this configuration c is locked. Obviously, this prevents the planner from selecting for any other $k \in \text{ActiveF}(p)$ a candidate configuration c for $\text{config}(k, p')$ which would cause interference. Note, here we just have to follow the edges of $\text{config}(f, p)$ to access the candidates potentially to be locked, cf. Fig. 3b.

5.3 (Permanently) Pinning Flows

To prevent the planner from ever reconfiguring an active flow f, we can *permanently* remove all configurations of f other than $\text{config}(f, p)$ from $\mathbf{G}(p)$, i.e., this *pins* an active flow to its configuration for its whole lifetime. Obviously, this ensures $\text{config}(f, p) = \text{config}(f, p')$ for future updates. In contrast, locking only temporarily excludes some candidates from p'.

6 Computing the New Traffic Plan

Once the new conflict-graph has been constructed by adding additional configurations and locking currently "forbidden" candidate configurations, we have to compute the traffic plan. From a graph-theoretic perspective, we can reduce the maximum (colorful) independent vertex set problem to the problem of finding a traffic plan which maximizes Eq. 2. This means, unsurprisingly, that computing a new traffic plan remains a NP-hard problem [25]. Therefore, we present a novel heuristic, namely, the *Greedy Flow Heap Heuristic* (GFH). The name draws from the fact that the objective Eq. 2 improves with every additional flow included in the new traffic plan. Next, we describe the GFH in detail, and explain our strategy to compute the new traffic plan.

6.1 Greedy Flow Heap Heuristic

From a birds-eye view, GFH is an iterative greedy approach that in each iteration computes ratings for the configurations in $\mathbf{G}(p')$ and adds the configurations with the best ratings to an intermediary set of conflict-free configurations $\mathcal{C}$. During the GFH execution $\mathcal{C}$ is always an independent vertex set in $\mathbf{G}(p')$. For the same flow f there may be multiple configurations in the final $\mathcal{C}$ returned by the GFH, but we can only include one of these (we need only one route and phase) in a traffic plan. In this case, we arbitrarily select one configuration for every flow with multiple configurations in $\mathcal{C}$ since all configurations in $\mathcal{C}$ are conflict-free.

Before explaining the greedy strategy and how configurations are rated, we introduce some terminology: $\mathcal{C}$ admits flow f, if it contains at least one candidate configuration for f. A configuration $c \in \mathbf{G}(p')$ is *shadowed* if at least one of its neighbors is included in $\mathcal{C}$. Adding a configuration to $\mathcal{C}$ shadows all its neighbors. Shadowed configurations are excluded from adding to $\mathcal{C}$ since they conflict, i.e., are connected by an edge, with at least one of the configurations in $\mathcal{C}$. A configuration is *solitary* if it has no neighbors in $c \in \mathbf{G}(p')$ and thus cannot conflict with any other configuration. A configuration c is *eligible* for selection by the GFH algorithm if and only if c is neither shadowed, nor $c \in \mathcal{C}$.

The GFH algorithm initializes $\mathcal{C}$ with the set of all solitary configurations. Thereby, $\mathcal{C}$ already admits all flows with solitary configurations without reducing the solution space. Then, configurations for the remaining flows are iteratively added to $\mathcal{C}$. If $\mathcal{C}$ does not admit all flows, GFH can be executed repeatedly, i.e., we say *re-run*, with different starting conditions in an attempt to improve the result. The number of re-runs (default $n_{\text{re-runs}} = 3$) can be parameterized.

6.1.1 Configuration Selection Strategy

GFH being a greedy algorithm, the quality of the solution, i.e., how many flows can ultimately be admitted, relies on the strategy we use to add configurations to $\mathcal{C}$. GFH uses a hierarchical, iterative strategy, first deciding on the next flow to process, and then selecting a configuration for that flow. Remember that any active flow contributes more to the objective than all new flows taken together. To prevent, that any configuration of a new flow shadows any active flow's configuration such that an active flow may not be admitted in $\mathcal{C}$, we thus first process all flows in $\text{ActiveF}(p)$, before searching a configuration for new flows from $\text{ReqF}(p)$.

The pseudo code to process a particular (sub-)set of flows, say $\text{searchF}(p') \subseteq \text{ActiveF}(p) \cup \text{ReqF}(p)$ is given in Alg. 2. We start by creating a min-heap that contains all flows from $\text{searchF}(p')$ not already admitted by $\mathcal{C}$ (cf. Alg. 2, line 2). The flows in the min-heap are sorted according to the number of remaining eligible configurations, i.e., flows with less eligible configurations are on top of the heap. Ties are broken with the higher total degree of the configurations in $\text{cand}(f, p')$, because a high total degree suggests a high chance that all configurations of the flow will soon become ineligible. The unique flow identifier is used as final tie-breaker to get a deterministic algorithm.

Next, the heap is iteratively processed: The flow on top of the heap—the flow with the least remaining eligible configurations—is removed from the heap, and we add the flow's best-rated configuration to $\mathcal{C}$. Thus, flows with few eligible configurations are processed first. When a configuration is added to $\mathcal{C}$, the heap is updated and may be re-ordered.

6.1.2 Rating Configurations

The shadowRating value of a configuration is intended to capture the "cost" of selecting that configuration in terms of the remaining solution space. For example, adding a configuration to $\mathcal{C}$

Algorithm 1: shadowRating

input : configuration c
shadowRating = 0 ;
$\mathcal{F}_{\text{neig}} \leftarrow$ flows of all neighbors of c;
foreach $f_{neig} \in \mathcal{F}_{neig}$ **do**
    shadowCount $\leftarrow$ number of *eligible* configurations of f_{neig} in neighborhood of v ;
    eligibleCount $\leftarrow$ total number of eligible configurations in f_{neig};
    $\delta \leftarrow$ shadowCount / eligibleCount ;
    if δ = *1* **then** shadowRating=shadowRating+α ;
    else shadowRating=shadowRating+δ ;
return shadowRating of c;

that shadows huge portions of the conflict graph is "expensive". Conversely, the cost is lower the less neighbors are shadowed. If neighbors are shadowed it is preferable to shadow configurations of those flows with lots of remaining eligible configurations.

Algorithm 2: addConfigPerFlow

input : flows set $\text{searchF}(p')$, conflict-free configurations $\mathcal{C}$
initialize empty *heap*;
$\forall f \in \text{searchF}(p')$: if **not** $\text{admits}(f, p')$ then insert f into *heap*;
while $heap \neq \emptyset$ **do**
    $f_{\text{min}} \leftarrow$ pop f_{min} with least eligible configurations from heap;
    $c_{\text{sel}} \leftarrow$ eligible c of color f_{min} with smallest $\text{shadowRating}(c)$;
    add c_{sel} to $\mathcal{C}$, update *heap* and shadowRating values;
return updated $\mathcal{C}$;

This intuition is encoded in the pseudo code given in Alg. 1: We compute for each flow f_{neig} in the 1-hop neighborhood of c the share δ of this flow's remaining eligible configurations that would be shadowed by picking c. Configurations which are already shadowed are not taken into account. The shadowRating ordinarily amounts to the sum over the δ values of c's neighborhood—except if adding c to $\mathcal{C}$ shadows all remaining eligible configurations of a flow ($\delta = 1$). To disincentive the selection of a configuration c which eclipses the remaining configuration of another flow, we add a very large constant α (default $\alpha = 1000$) instead of 1 to the shadowRating value of c. Note that the shadowRatings can be computed in parallel, since we only have to read the neighbors of c.

To give a brief example how GFH works, we revisit the conflict graph given in Fig. 3b. As there are no solitary configurations GFH starts with $\mathcal{C} = \emptyset$. The active flows f_a and f_b are handled by the first addConfigPerFlow call. Hence, the heap consists of these two flows where f_a has five and f_b has three eligible configurations, respectively. First, the best configuration of flow f_b is calculated since f_b has less eligible configurations. Since $b1$ shadows all configurations of f_c, $b1$ gets a very high shadowRating whereas $b2$ gets the best rating and is added to $\mathcal{C}$, since it only shadows $a2$. Second, flow f_a is processed. Here, $a3$ has the lowest shadowRating ($a4$ and $a5$ are locked) and is added to $\mathcal{C}$. Finally, addConfigPerFlow is called the second time with $\text{searchF}(p') = \{f_c\}$ since f_c is the only new flow to add. Thereby, $c1$, which the only configuration of f_c, is added to $\mathcal{C}$.

6.1.3 Re-run Mechanism

The re-run mechanism can be used to improve the number of flows admitted by $\mathcal{C}$. Remember that the order of the flow sub-sets for which we execute addConfigPerFlow implicitly assigns priorities to those subsets. Therefore, for each re-run we split both ActiveF(p) and ReqF(p) into two sub-sets each: one subset contains all flows which were previously admitted by $\mathcal{C}$, and another subset contains those flows which could not be admitted previously.

Flows which are not admitted by the previous $\mathcal{C}$ seemed to be particularly "problematic" in the previous run. Hence, this time, we invoke addConfigPerFlow for those flows first. As a result, we call addConfigPerFlow on the four flow sub-sets in the following order: 1) previously not by $\mathcal{C}$ admitted flows from ActiveF(p), 2) previously by $\mathcal{C}$ admitted flows from ActiveF(p), and 3), 4) equivalently for flows in ReqF(p). This can improve the objective value, but it is not guaranteed. Re-runs are performed either until $\mathcal{C}$ admits all flows, or we run out of re-runs. In the latter case, the GFH algorithm returns the $\mathcal{C}$ with the highest objective value seen so far.

6.2 Rejecting and Removing Flows

If ReqF(p) contains flows that are not admitted by $\mathcal{C}$, the planner rejects the corresponding flows. All candidate configurations for a rejected flow are consequently purged from $\mathbf{G}(p')$. Similarly, applications could indicate to the planner that active flows shall be removed from the network, in which case the planner also purges the corresponding candidate configurations from the conflict graph.

6.3 Optimization: Progressive Strategy for Offensive Planning

The planner employs locking and GFH in a two-phase meta-strategy for offensive planning:

In the first, defensive phase, for each flow in ActiveF(p) *every* configuration in $\text{cand}(f, p') \setminus \text{config}(f, p)$ is locked in $\mathbf{G}(p')$, i.e., we conserve the configuration of all active flows. Then $\mathbf{G}(p')$ exposes only one configuration per active flow and all candidates for new flows to the GFH. If we already find a traffic plan p' for $\text{ActiveF}(p) \cup \text{ReqF}(p)$ we are done, and usually saved computation time since the GFH considers fewer configurations.

Only if we cannot admit all new flows, we lift the conservative locks (configurations which cause transition interference, cf. Sec. 5.2 remain locked) and expose the "full" conflict-graph to the GFH. This second phase widens the search space for the GFH at the expense of longer runtime, and active flows now may be reconfigured. If the GFH rejects any active flows in the second phase, we revert to the result from the first phase which is guaranteed to include all active flows in the new traffic plan p'.

7 Installing the Plan

After having computed the new plan, the controller must propagate sub-plans to the nodes in the network. The sub-plan sent to a particular infrastructure node defines how the nodes are supposed to route incoming packets, when to reserve transmissions windows for these packets, and when the new plan becomes active, i.e., the traffic plan's activation time t_{act}.

Obviously, to make the nodes to agree on the new traffic plan and its activation time amounts to a consensus problem. For the sake of simplicity, we will assume here that neither controller nor nodes fail. Otherwise, a consensus protocol is needed taking into account those failures, and which takes special action if consensus cannot be achieved before the new plan's activation time. Those protocols are beyond the scope of this paper.

Here, we assume that the controller has a real-time control channel to each network node. These channels are established when the network is initialized, and they are not subject to reconfigurations. Since the configurations of control channels are fixed, e.g., implemented with pinning, their associated QoS is not subject to degradation. With real-time channels and an upper-bound on the time between receiving a new sub-plan and the time it takes the node to enact it, i.e., how long it takes the node to update routing entries and transmission schedules, the controller can set the activation time to a hyper-cycle boundary in the future. Pushing the activation time farther into the future allows alternative control channel implementations with higher worst-case delay bounds than time-triggered flows. For instance, in networks with TAS, we could use a separate traffic class with bandwidth guarantees.

As discussed in Sec. 5, the controller has to postpone the activation time for *sources* of new flows after the transition interval whereas this is not necessary (and it would also be impractical) for infrastructure nodes that forward packets of new flows. Postponing the activation of a new source practically means to add an offset of $\lceil \frac{t_{\text{transit}}}{t_{\text{cycle}}} \rceil \cdot t_{\text{cycle}}$ for each new flow to the activation time, where t_{transit} is the duration of the longest transition interval.

8 QoS Considerations

The reconfiguration of active flows can degrade the QoS by introducing jitter. Since we aim for deterministic real-time communication, we must also quantify and possibly contain the QoS degradation caused by reconfigurations to a level acceptable by the applications. Next, we study how the reconfiguration of an active flow can degrade QoS. The level of degradation depends on the "distance" of the flow's old and new configuration, i.e., the phase shift and the difference in the lengths of the routes. After showing how these properties affect QoS degradation, we present a way how applications can control the degree of degradation.

8.1 Computing QoS Degradation

Without reconfiguration, the destination node of an active flow receives the next packet every t_{cycle} seconds after the reception of the previous packet, and packets are received in the order they were sent from the source node. Now assume, a new traffic plan p' supersedes p, and an active flow $f \in \text{ActiveF(p)}$ is reconfigured, i.e., $\text{config}(f,p) \neq \text{config}(f,p')$. W.lo.g., the activation time of the new traffic plan p' is the start of the $n+1$-th cycle of f. Consequently, the point in time t'_{rx} when the destination node receives the $n+1$-th packet, which is sent with $\text{config}(f,p')$, may differ from the point in time t_{rx} when the destination would have received the $n+1$-th packet without reconfiguration, i.e., if the $n+1$-th packet would have been sent with the old $\text{config}(f,p)$, cf. Fig 4. We call the magnitude of this deviation $\Delta t = t'_{\text{rx}} - t_{\text{rx}}$ the (reconfiguration) *jitter*. Δt can be computed with the following theorem.

Theorem 1 (Δt)**.** *After a reconfiguration of flow f from $(f,\phi,\pi) = \text{config}(f,p)$ to $(f,\phi',\pi') = \text{config}(f,p')$, the arrival of the new packets at the destination node deviates by $\Delta t = (\phi' - \phi) + (l(\pi') - l(\pi)) \cdot t_{perhop}$ from their respective scheduled arrival according to previous config(f,p) with $l(*)$ denoting the number of infrastructure nodes on the candidate path with index $*$.*

Proof. W.lo.g. we use the start n-th cycle as reference time t_{ref}. In the n-th cycle, the source node starts transmitting the n-th packet at $t_{\text{ref}} + \phi$. The reception of this packet by the destination node starts at $t_{\text{ref}} + \phi + t_{\text{src}} + t_{\text{trans}} + t_{\text{prop}} + l(\pi) \cdot t_{\text{perhop}} + t_{\text{dst}}$ (cf. Sec. 3.4). If p remained valid, the

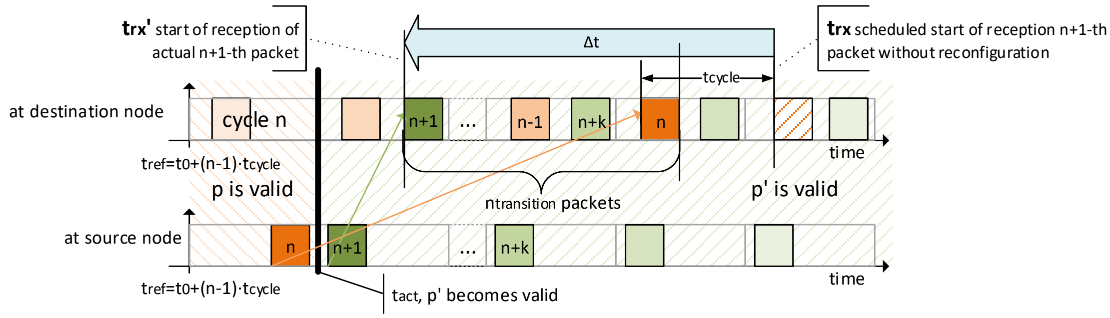


Figure 4: Reconfiguration of an active flow can temporarily cause jitter and packets re-ordering.

reception of the $n+1$-th packet then would start t_{cycle} later at $t_{\text{rx}} = t_{\text{ref}} + \phi + t_{\text{src}} + t_{\text{trans}} + t_{\text{prop}} + l(\pi) \cdot t_{\text{perhop}} + t_{\text{dst}} + t_{\text{cycle}}$.

However, the new plan p' and thus config(f, p') becomes valid at $t_{\text{ref}} + t_{\text{cycle}}$, which is the start of the $n+1$-th cycle. The reception of the $n+1$-th packet sent by the source node with config(f, p') start at $t'_{\text{rx}} = t_{\text{ref}} + t_{\text{cycle}} + \phi' + t_{\text{src}} + t_{\text{trans}} + t_{\text{prop}} + l(\pi') \cdot t_{\text{perhop}} + t_{\text{dst}}$ at the destination node. By inserting these values, we get $\Delta t = t'_{\text{rx}} - t_{\text{rx}} = (\phi' - \phi) + (l(\pi') - l(\pi)) \cdot t_{\text{perhop}}$. □

If $\Delta t > 0$, there is an additional delay between the last packet of $\langle f, p\rangle$ and the first packet of $\langle f, p'\rangle$. If $\Delta t < 0$, the new packets from $\langle f, p'\rangle$ arrive "too early". Potentially, this can lead to a situation, where packets from $\langle f, p'\rangle$ overtake the last packets from $\langle f, p\rangle$ in the network if the relative end-to-end deadline is allowed to be greater than the cycle time. We can upper bound the number of packets which possibly arrive in a different order than the order in which they were sent, and do not have an inter-arrival time t_{cycle} by $n_{\text{transition}} \leq 2 \cdot \left\lceil \frac{|\Delta t|}{t_{\text{cycle}}} \right\rceil$.

8.2 Restricting QoS Degradation

Applications can provide the planner with QoS degradation constraints in the form of thresholds on $|\Delta t|$ and the number affected packets. If the destination node has no facilities to handle packet reorderings, e.g., has no buffer, then such constraints can be used to ensure that all packets are received in the send order. The QoS degradation constraint results in the following condition for $\mathbf{G}(p')$:

QoS condition For each $f \in \text{ActiveF}(p)$, there exists no configuration $c \in \text{cand}(f, p')$, where the reconfiguration from config(f, p) to c exceeds the threshold on $|\Delta t|$.

Analog to Condition 1 from Sec. 5.2, we can prevent reconfigurations that violate the QoS condition via *locking*. This can be achieved with minimal overhead since the planner anyway traverses $c \in \text{cand}(f, p')$, and only needs to check the QoS condition for each candidate c of f that is not locked already due to transition interference.

9 Evaluation

We evaluated a prototypical C++-implementation of the planner with pre-generated flow-request sequences. Starting with the previous traffic plan p and ActiveF(p), the planner processes ReqF(p)

Table 1: Overview of evaluation scenario parameters.

Fig.	|ActiveF(p)|	|ReqF(p)|	flows to remove	n_{ub}	t_{cycle} [µs]	n_{path}	network
5a	500	25 (poiss.)	25 (poiss.)	$\leq$ {50,100}	{250,500,1000,2000}	3	ring(64,3)
5b,5c	800	50 (poiss.)	50 (poiss.)	$\leq$ {50,100}	{250,500,1000,2000}	3	ring(64,3)
6	250	25 (det.)	25 (det.)	$\leq$ 100	{200,250,500}	3	ring(64,3)
7a, 7b	500	50 (det.)	50 (det.)	$\leq$ 100	{250,500}	5	var., 49 nodes
7a, 7c	500	50 (det.)	50 (det.)	$\leq$ 100	{250,500}	5	var., 81 nodes

t_{trans} (packet size)	$\in \{1\,\mu s, 3\,\mu s, 5\,\mu s, 12\,\mu s\}$, i.e., packets sizes: 125 B – 1500 B on links with 1 Gbit s^{-1}
flow clusters	$\in \{1, 2, 4, 8, 16, 32\}$ (constrained by |ReqF(p)| and amount of flows to remove)
network parameters	processing delay $t_{\text{proc}} = 2\,\mu s$, propagation delay $t_{\text{prop}} = 1\,\mu s$

and requests for removal of active flows from ActiveF(p) in one step and validates the absence of configuration-conflicts in the new traffic plan. By default, we used a desktop-grade computer with Intel i7-10700K (8 cores) and 16 GB RAM for the evaluations.

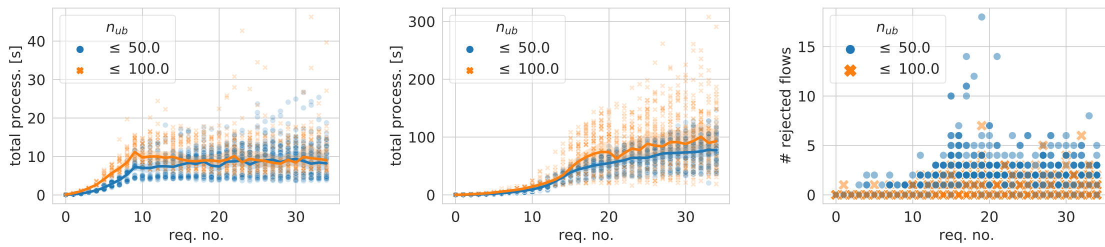
Tab. 1 gives an overview over the evaluation parameters which are derived from typical industrial use cases. For each flow, we draw the values for t_{trans} and t_{cycle} uniformly at random from the respective set. The number of new flows to add, i.e., |ReqF(p)| and the number of flows to remove is either drawn from a truncated Poisson distribution (poiss.) with averages as stated in Tab. 1, or is deterministic (det.) for each processing step. We place/remove the flows in clusters in the network to simulate control units connected to multiple sensors and actuators, i.e., all flows in a cluster start or target one common "cluster" node. For example, to add 25 new flows the request may contain three clusters off size 1, 8 and 16 flows, respectively, or any other combination that adds up to 25 flows. By default, 20% of the new flows are pinned permanently, and we limit Δt for each remaining active flow to $t_{\text{cycle}} - t_{\text{trans}}$ such that packets arrive in order at the destination node.

In Tab. 1, column n_{ub} denotes the limit of the number of candidate configurations the planner may add for each flow to $\mathbf{G}(p')$, and n_{path} is the upper bound of candidate paths per flow. In column *network*, ring(n, k) denotes a circular network with n nodes where each node is connected to the next k nodes in both directions (resulting in equal node-degrees and equal number of alternative paths between any pair of nodes).

9.1 Runtime

We evaluated 60 scenarios wrt. runtime for each combination of average active flows (500, 800) and per-flow candidate set increment limit (50, 100). Fig. 5 shows the results for 35 processing steps, and we measured up to 9.072 GB RAM usage by the planner. We plot the *total* runtime for each processing step in Fig 5a and 5b which includes the time to compute the candidate paths for new flows, all the operations on the conflict-graph (adding, locking, pinning and removing candidate configurations) and the GFH runtime. We use the first 10 (16) processing steps to initialize the conflict graph from scratch by requesting only new flows to be added until ActiveF(p) contains 500 (800) flows. During these initialization steps, we observe an increasing total runtime. In Fig. 5a the runtime per step plateaus (mean: 8.24 s, n_{ub}=50; 9.2 s, n_{ub}=100) after the initialization steps. Here, the conflict graph size remains "stable", i.e., the planner effectively exchanges 25 active flows (and their configurations) against 25 new flows (and the respective new configurations).

In Fig. 5b, the runtime continues to grow after the initial 16 steps (max: 146.83 s for n_{ub}=50;



(a) Total runtime per processing step (±500 active flows).

(b) Total runtime per processing step (±800 active flows).

(c) Rejected new flows per step (±800 active flows).

Figure 5: Runtimes and number of rejected flows per processing step (500 and 800 active flows).

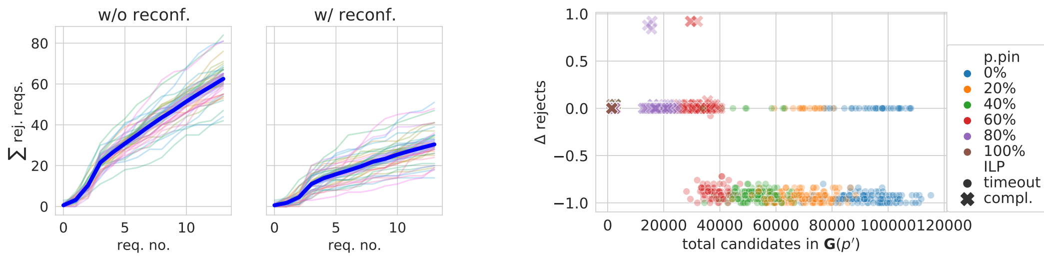
307.72 s for n_{ub}=100). This behavior can be explained with Fig. 5c. Fig. 5c is a scatter plot of the number of "rejected" new flows for the scenarios with 800 flows, i.e., we plot for each scenario how many new flows the planner did not admit in new traffic plan. As discussed in Sec. 5.1, the planner adds more candidates for active flows after the first pass over the ϕ-π-space if flows from the previous request had to be rejected. This feedback loop causes the planner to grow the conflict graph more aggressively. The effects of varying how many additional candidates the planner can add at most to the conflict graph for each can also be observed: if the planner may add up to 100 additional candidates per flow it can admit more new flows, but in turn we observe generally longer runtimes. In the scenarios with 800 flows the planner rejected on average 0.84 (0.23) flows per request after the initialization steps for $n_{\mathrm{ub}} = 50$ ($n_{\mathrm{ub}} = 100$) per step.

9.2 Comparison

Next, we compare defensive and offensive traffic planning wrt. the network utilization, and show that new algorithms such as the GFH are required to make offensive traffic planning feasible. The evaluation scenarios vary with respect to how many flows are permanently pinned (abbr. perm. pin) on average to their (initial) configuration. We increase the ratio of pinned flows from 0% to 100% in 20%-increments and evaluated 40 scenarios for each step. Here, there is no QoS restriction on Δt for each active flow that can be reconfigured.

In Fig. 6b, we plot the cumulative number of flows rejected by the planner over the 14 processing steps. By pinning every active flow (cf. w/o reconfiguration, permanently pinning 100% of flows in Fig. 6), our approach performs defensive traffic planning, which results in a total of 62.5 rejected new flows on average. In comparison, if the planner performs offensive traffic planning less than half as many flows —30.4 flows on average—are rejected (cf. w/ reconfiguration, permanently pinning 0% of flows in Fig. 6).

Next, we show that the GFH algorithm pushes the boundary of scenarios that can be tackled with offensive traffic planning far beyond what is possible with state-of-the-art methods. To also answer, how "optimal" the GFH results are, we compare GFH against an ILP solver since Eq. 2 and the required conflict-freeness for candidates in $\mathbf{G}(p')$ can be expressed with linear expressions. We saved the conflict graph instances from the last four processing steps from Fig. 6b—presumably the largest conflict graphs in each scenario—to disk, and solved the corresponding ILP instances with a tool-chain using Julia [26], JuMP [27] and Gurobi v9.1.1 [28]. The ILP solver had a runtime limit of 5 min (GFH max=8.3 s), and GFH and ILP solver were limited to using 16 threads. Due to the higher



(a) Cumulative rejected flows (thin lines: per scenario, thick line: average).

(b) Frequent timeouts of the ILP solver for larger conflict graphs result in worse results than GFH.

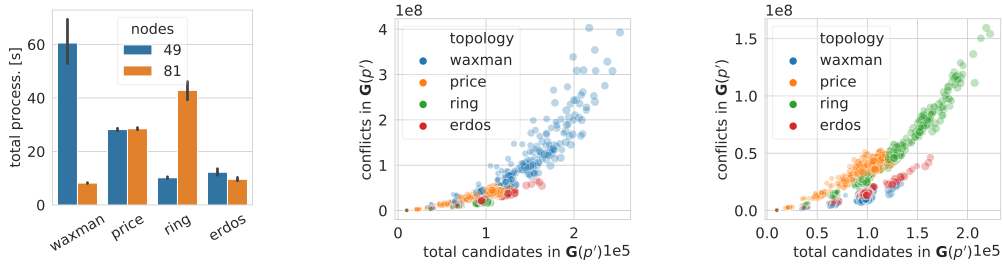
Figure 6: Flow reconfiguration can improve the number of admitted new flows, but results in large conflict graphs which can only be processed with novel algorithms such as GFH.

memory requirements (up to 56.694 GB) of the ILP tool chain, we used a server-grade computer with two AMD EPYC 7401 processors (each 24 cores) and 256 GB RAM. Fig. 6a plots for each processing step the relative difference in the number of rejected new flows $\Delta\text{rejects} = \frac{\text{rejects GFH} - \text{rejects ILP}}{\text{new flows (total)}}$ over the number of candidate configurations in the conflict graph. If the ILP solver rejected fewer flows, i.e., computed a better result we have $\Delta\text{rejects} > 0$, and vice-versa. In Fig. 6a, the ILP solver could provide better solutions for small conflict graphs, and, except for a few outliers where GFH reverted to the result from the first phase, the advantage of the ILP over GFH is small. Yet, once conflict graphs contain 30'000 or more candidate configurations, the ILP solver frequently hits the time-limit (for 0% pinned flows: GFH avg=4.9 s±1.8 s; ILP avg=290.7 s±71.4 s) and cannot produce a traffic plan which admits many new flows. This means, that offensive traffic planning is limited to either small scenarios or a small fraction of the flows (cf. pinning 80% of all flows) without the GFH algorithm.

9.3 Network Topology

To investigate the effects of network topology and network size, we generated scenarios for networks with 49 nodes and 81 nodes for different graph models, namely, Waxman, Price, ring(n, k), and Erdős-Rényi, using graph generators from [29, 30] We evaluated 60 scenarios for each size-topology combination with active 500 flows on average. After the initialization the planner tries adding 50 new flows and removing 50 active flows in each step.

The average total runtimes after initialization and warm-up steps are plotted in Fig. 7a. We observe that a larger network does not per-se result in higher runtimes or bigger conflict graphs, cf. Fig 7b, 7c. Waxman networks with 49 nodes result in the largest conflict graph sizes overall and highest average (60.5 s) and maximal (283.72 s) total runtime per step. In Price networks with only one path between every two nodes, the planner cannot resolve conflicts via routing. Compared to the other topologies, on average in more than ten times as many new flows were rejected per step (49n. avg=10.4 rej./step; 81n. avg=11.6 rej./step) in scenarios with Price network topologies. In scenarios with Price networks, GFH often required all re-runs, and we measured the third-highest total runtimes.



(a) Per-step total runtime. (b) Conflict graph (49 nodes). (c) Conflict graph (81 nodes).

Figure 7: Impact of different network topologies on total runtime and conflict graph size.

10 Conclusion and Outlook

In this paper, we presented a novel approach for offensive dynamic traffic planning, which allows to reconfigure active flows to achieve better network utilization, and a novel algorithm for computing these traffic plans. We are able to quantify and control the QoS degradation a flow may suffer during such reconfigurations. Our evaluations show that we can efficiently compute updated traffic plans for scenarios with hundreds of active flows.

Interesting directions for future work include investigating memory-access optimized conflict graph data-structures and the relaxation of the zero-queuing constraint, as well as protocols and concepts for error-handling during the traffic plan deployment.

References

[1] IEEE Computer Society, "IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks," *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)*, pp. 1–1993, 2018.

[2] N. Finn and P. Thubert, "Deterministic Networking Problem Statement," RFC 8557, May 2019. [Online]. Available: https://rfc-editor.org/rfc/rfc8557.txt

[3] F. Dürr and N. G. Nayak, "No-wait Packet Scheduling for IEEE Time-sensitive Networks (TSN)," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, ser. RTNS '16. ACM, 2016, pp. 203–212. [Online]. Available: http://doi.acm.org/10.1145/2997465.2997494

[4] P. Danielis, H. Puttnies, E. Schweissguth, and D. Timmermann, "Real- Time Capable Internet Technologies for Wired Communication in the Industrial IoT-a Survey," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2018, pp. 266–273.

[5] G. Karagiannis, O. Altintas, E. Ekici, G. Heijenk, B. Jarupan, K. Lin, and T. Weil, "Vehicular Networking: A Survey and Tutorial on Requirements, Architectures, Challenges, Standards and Solutions," *IEEE Communications Surveys Tutorials*, vol. 13, no. 4, pp. 584–616, 2011.

[6] O. Sadio, I. Ngom, and C. Lishou, “Design and Prototyping of a Software Defined Vehicular Networking,” *IEEE Transactions on Vehicular Technology*, vol. 69, no. 1, pp. 842–850, 2020.

[7] E. Schweissguth, P. Danielis, D. Timmermann, H. Parzyjegla, and G. Mühl, “ILP-based Joint Routing and Scheduling for Time-triggered Networks,” in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, ser. RTNS ’17, 2017, pp. 8–17. [Online]. Available: http://doi.acm.org/10.1145/3139258.3139289

[8] W. Steiner, S. S. Craciunas, and R. S. Oliver, “Traffic Planning for Time-Sensitive Communication,” *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 42–47, 2018.

[9] M. Pahlevan, N. Tabassam, and R. Obermaisser, “Heuristic list scheduler for time triggered traffic in time sensitive networks,” *ACM SIGBED Review*, vol. 16, no. 1, pp. 15–20, 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3314206.3314208

[10] N. G. Nayak, F. Dürr, and K. Rothermel, “Incremental Flow Scheduling and Routing in Time-Sensitive Software-Defined Networks,” *IEEE Transactions on Industrial Informatics*, vol. 14, no. 5, pp. 2066–2075, 2018.

[11] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, “Fastpass: A Centralized ”Zero-queue” Datacenter Network,” in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM ’14. ACM, 2014, pp. 307–318. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626309

[12] J. Falk, F. Dürr, and K. Rothermel, “Time-Triggered Traffic Planning for Data Networks with Conflict Graphs,” in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 124–136.

[13] A. Atallah, G. Bany Hamad, and O. Ait Mohamed, “Routing and Scheduling of Time-Triggered Traffic in Time Sensitive Networks,” *IEEE Transactions on Industrial Informatics*, pp. 1–1, 2019.

[14] J. Falk, F. Dürr, and K. Rothermel, “Exploring Practical Limitations of Joint Routing and Scheduling for TSN with ILP,” in *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2018, pp. 136–146.

[15] N. G. Nayak, F. Dürr, and K. Rothermel, “Time-sensitive Software-defined Network (TSSDN) for Real-time Applications,” in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, ser. RTNS ’16. ACM, 2016, pp. 193–202. [Online]. Available: http://doi.acm.org/10.1145/2997465.2997487

[16] F. Smirnov, M. Glaß, F. Reimann, and J. Teich, “Optimizing Message Routing and Scheduling in Automotive Mixed-Criticality Time-Triggered Networks,” in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC ’17. ACM, 2017, pp. 48:1–48:6. [Online]. Available: http://doi.acm.org/10.1145/3061639.3062298

[17] S. S. Craciunas and R. S. Oliver, “Combined task- and network-level scheduling for distributed time-triggered systems,” *Real-Time Systems*, vol. 52, no. 2, pp. 161–200, 2016. [Online]. Available: http://link.springer.com/10.1007/s11241-015-9244-x

[18] R. S. Oliver, S. S. Craciunas, and W. Steiner, "IEEE 802.1Qbv Gate Control List Synthesis Using Array Theory Encoding," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018, pp. 13–24.

[19] F. Pozo, G. Rodriguez-Navas, W. Steiner, and H. Hansson, "Period-Aware Segmented Synthesis of Schedules for Multi-hop Time-Triggered Networks," in *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2016, pp. 170–175.

[20] W. Steiner, "An Evaluation of SMT-Based Schedule Synthesis for Time-Triggered Multi-hop Networks," in *2010 31st IEEE Real-Time Systems Symposium*, 2010, pp. 375–384.

[21] A. Syed and G. Fohler, "Efficient offline scheduling of task-sets with complex constraints on large distributed time-triggered systems," *Real-Time Systems*, vol. 55, no. 2, pp. 209–247, 2019. [Online]. Available: https://doi.org/10.1007/s11241-018-9320-0

[22] R. S. Oliver, S. S. Craciunas, and G. Stöger, "Analysis of Deterministic Ethernet scheduling for the Industrial Internet of Things," in *2014 IEEE 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, 2014, pp. 320–324.

[23] J. Y. Yen, "Finding the K Shortest Loopless Paths in a Network," *Management Science*, vol. 17, no. 11, pp. 712–716, 1971.

[24] ISO, IEC, and IEEE, "ISO/IEC/IEEE International Standard – Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements – Part 1Q: Bridges and bridged networks AMENDMENT 1: Path control and reservation," pp. 1–122, 2017.

[25] M. R. Garey and D. S. Johnson, *Computers and Intractability*, ser. A Guide to the Theory of NP-Completeness. Freeman, 1979.

[26] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017. [Online]. Available: https://doi.org/10.1137/141000671

[27] I. Dunning, J. Huchette, and M. Lubin, "Jump: A modeling language for mathematical optimization," *SIAM Review*, vol. 59, no. 2, pp. 295–320, 2017.

[28] L. Gurobi Optimization, "Gurobi optimizer reference manual," 2020. [Online]. Available: http://www.gurobi.com

[29] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[30] T. P. Peixoto, "The graph-tool python library," *figshare*, 2014. [Online]. Available: http://figshare.com/articles/graph_tool/1164194